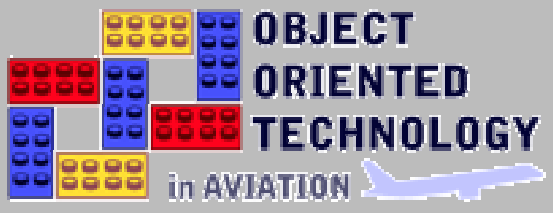
DRAFT

Handbook for Object-Oriented Technology in Aviation (OOTiA)

Volume 2: Considerations and Issues

vPC.0

January 30, 2004



This Handbook does not constitute Federal Aviation Administration (FAA) policy or guidance, nor is it intended to be an endorsement of object-oriented technology (OOT). This Handbook is not to be used as a standalone product but, rather, as input when considering issues in a project-specific context.

Contents

Figures

2.1 Introduction

The introduction of object oriented techniques and tools to aviation software development presents challenges to understanding their effect on safety and certification. As discussed in Volume 1, there is an increasing desire among aviation software developers to use object-oriented technology (OOT), including object oriented modeling, design, programming, and analysis, in the development of aviation applications. These desires are fueled, at least in part, by claims of increased efficiency in the development of complex systems through using reusable components. Object oriented (OO) design, with the ability to encapsulate design decisions, is considered by some to be "the most important low-level design technology in modern software engineering" [19].

In response to the aviation industry's desire to use OOT, the Federal Aviation Administration (FAA) enlisted the National Aeronautics and Space Administration (NASA) to help start the Object Oriented Technology in Aviation (OOTiA) project. This project is sponsoring research and conducting workshops designed to identify concerns about OOT relevant to safety and certification and to develop recommendations for its safe, and DO-178B compliant, use.

The OOTiA project was initially based on work by the Aerospace Vehicle Systems Institute (AVSI). AVSI is a research consortium for the aerospace industry working to reduce the costs of complex subsystems in aircraft. As part of this consortium, Boeing, Honeywell, Goodrich, and Rockwell Collins collaborated on an AVSI project titled *Certification Issues for Embedded Object-Oriented Software*, the goal of which was to mitigate the risk that individual projects face when certifying systems with OO software. The AVSI project proposed a number of guidelines for producing object-oriented DO-178B compliant software [2].

In 2001, a committee including representatives from the AVSI project, FAA, and NASA Langley Research Center, was formed to extend the AVSI work for the benefit of the entire aviation software community. This committee developed the following approach for accomplishing this purpose:

- Establish a web site dedicated to collecting data on safety and certification concerns
- Hold public workshops to which the aviation software community would be invited to discuss concerns
- Document each key concern raised either through the web site or the workshops
- Adapt the AVSI guidelines to address the concerns
- Produce a handbook.

This report is the second volume of the OOTiA handbook. This volume focuses expressly on the concerns and questions about OOT that have been collected through the OOTiA web site and workshops, with the goal of raising awareness of aspects of OOT that may complicate compliance with DO-178B. Consequently, the tone of this report may seem overly negative to some readers, just as the tone of other volumes may seem overly positive to others. Readers of this report should carefully note the following:

- Comments recorded through the OOTiA activities are cited throughout the report. The purpose of citing recorded comments is not to imply their individual validity, but to account for the data that has been collected and show the basis for a concise set of key concerns relevant to DO-178B compliance that are derived from the data *as a whole*.
- The key concerns documented in this report do not constitute a complete set of safety and certification concerns.
- Nearly all of the submitted issues used a particular nomenclature for OO concepts and constructs (class, subclass, superclass, method). For simplicity, the same nomenclature is used in this volume. This usage does not imply a preference for languages or tools that use these terms over those that do not.
- This volume does not discuss approaches for how to resolve the concerns. Other volumes of the handbook (Volume 3 in particular) are purported to provide resolutions.

Note also that this volume assumes that the reader has a fundamental understanding of OOT concepts and languages. Further information and references on these can be found in Volume 1.

2.1.1 Background

On September 14, 2001, the OOTiA web site http://shemesh.larc.nasa.gov/foot/ was launched by NASA Langley Research Center, and the aviation software community was invited by email to participate in a dialogue about OOT. The email distribution list comprised over 900 individuals who had expressed an interest in or attended software related functions sponsored by the FAA. Individuals were invited to participate by submitting comments or concerns about OOT to an issue list kept on the OOTiA web site, by attending public workshops organized by the OOTiA committee, and by reviewing products from the effort.

To date, 103 separate concerns[1] about various aspects of OOT have been collected through the web site. The web site initially requested that each submission include a topic, a statement of the concern, and a proposed solution (if known). Neither individual nor company names were recorded with the submittals. No specific guidance was given regarding what could or could not be submitted. Later updates of the web site simply requested that concerns be emailed to a point of contact at NASA Langley. The web site continues to accept new submissions.

Each submission to the web site is added to a list titled "Issues and Comments about Object Oriented Technology in Aviation." This issue list is posted on the web site and updated as new issues are submitted. Every entry that is submitted is added to the list exactly as it is submitted; entries to the list are not edited. As of the date of publication of this report, the web site is operational. Decisions about how to respond to future submissions or when to close the web site have not been made.

Considerable overlap and similarities are evident when reviewing the entries in the issue list. The OOTiA committee originally determined that most of the issues on the issue list related to the following eight topics: single inheritance, multiple inheritance, reuse and dead/deactivated code, tools, templates, overloading, type conversion, and inlining. The OOTiA committee drafted papers for each of these topics, drawing heavily from the original AVSI documents.

In April 2002, a public workshop was held to introduce the OOTiA project, to discuss the draft papers, and to provide an opportunity for people to raise additional concerns about OOT. Workshop attendance included 13 FAA representatives and more than 100 aviation industry representatives supporting both airborne and ground-based applications. After this workshop, the individual draft papers were revised and collated into a single document: "Handbook for Object Oriented Technology in Aviation." Also, a ninth topic, traceability, was added, and a paper on the topic included in the draft handbook.

The draft handbook served as the basis for discussion at a second public workshop, held in March 2003. Attendance at this workshop was similar in number and composition to the first workshop. Results of both workshops are available on the OOTiA web site. Most of the workshop was devoted to individual sessions on specific chapters of the handbook. The purpose of those sessions was to review and modify the draft chapters, and to document new issues, if raised. Two other sessions at the workshop were not directly tied to the handbook: Beyond the Handbook and Open Issues. The Beyond the Handbook session provided participants with an opportunity to discuss questions that should be answered *before* making a decision to use OOT on a project. The Open Issues session provided participants the opportunity to discuss any concerns they thought had not been adequately covered in the draft handbook.

2.1.2 Purpose and Organization of Volume 2

"Change in general rarely comes without cost, and change in one area often raises new challenges in other areas" [17]. The purpose of this document is to report and discuss the challenges collected throughout the OOTiA program. The discussion is organized as follows. In section 2.2, results of the Beyond the Handbook session are presented as issues to be considered before the decision to use OOT has been made. Section 2.3 presents issues documented about OOT that should be considered after the decision to use OOT has been made. This discussion focuses on the entries to the issue list, and identifies key concerns with respect to the life cycle processes specified in DO-178B. Section 2.4 reports those items that have been proposed as open issues. Finally, section 2.5 summarizes the challenges and discusses their relationship to other volumes of the handbook.

[1] There are actually 107 entries to the list, but 4 of them are duplicates.

2.2 Considerations Before Making the Decision to Use OOT

Much of the focus of the OOTiA program has been on *how to use* OOT, assuming that the decision to use OOT on a program has already been made. At OOTiA Workshop 2, participants in the Beyond the Handbook session were asked to brainstorm questions that should be answered *before* a program commits to using OOT. During the session, participants produced a list of fifty-one questions related to making a decision about whether to use OOT. At the end of the brainstorming session, these questions were reviewed, grouped according to the scheme shown in Figure 1, and presented to the plenary session at Workshop 2.

1 Making the choice for OOT
How to determine whether OOT is the correct approach for producing a solution with the appropriate level of integrity for the problem

1.1 OOT Metrics
Where is the body of knowledge that shows that OOT allows us to build systems with all of the benefits that have been claimed for it; e.g., is OOT better, cheaper, faster, safer, more reusable, more maintainable, more reliable, etc?

1.2 Technical Challenges
How to determine the technical risk involved in applying OOT to ensure the appropriate level of integrity? Methods for determining and mitigating risk include research in the academic and corporate communities, corporate experience, and expert knowledge.

1.3 Available resources
What resources, specific to adopting OOT, are needed to ensure the appropriate level of integrity? Examples of resources include skills, training, processes, corporate experience, tools, standards, computing capacity, time, and money. On the subject of certification, there are 2 needed components: guidance and skills.

2 DO-178 issue
Are all the objectives (Annex A) in DO-178B compatible with OOT? Is there anything specific to OOT that is not addressed in DO-178B?

3 Handbook Issues
How should the handbook be applied to a practical project? For example, is it a certification aid or best practice? Is it adequate – that is, what needs to be extended, added, and changed?

Figure 2.2-1 Original Classification Scheme for the Beyond the Handbook Questions

Appendix A contains the original questions recorded during the brainstorming session. Since that time, the questions from the session have been re-examined and placed in one of the following five categories:

- Reality of the Benefits
- Project Characteristics
- OOT Specific Resources
- Regulatory Guidance
- Technical Challenges

The rest of this section briefly discusses each of these five categories and the key questions and associated issues within each.

2.2.1 Reality of Benefits

The first question that should be asked and answered is:

(1) *What are the benefits of OOT compared with current or alternative approaches? And, what evidence exists to support claimed benefits of better, cheaper, faster, safer, more reliable, more maintainable, etc.?*

In a recent article on object orientation [13], Robert Glass stated that "[t]he software field has been subjected, over the years, to excessive claims of benefits for almost every new technology." OOT is no exception. OOT has become a popular software development approach within many non-safety-critical industries, due in part to claims related to reuse and associated benefits of efficient development. Within a group of aviation software engineers, it is not surprising that questions were raised about evidence to support or deny such claims. Participants in this session were particularly concerned about finding evidence to support extrapolating advantages claimed for OOT in non-safety critical systems to safety-critical systems. Because OOT has been around for a relatively long time, one would think there would be an abundance of evidence to promote thorough understanding of OOT benefits. There is an abundance of information, but how much of it qualifies as credible evidence relevant to the aviation software industry is debatable.

Studies exist that support the claimed benefits, such as Basili's results showing reduced defect density and rework with OOT [4], while other studies show potential problems such as inferior understandability, complexity, and maintenance problems [34, 31, 25]. Most studies, these included, are open to criticism, both about internal validity (did the experimental treatments really make a difference?) and external validity (to what populations and settings may the results be generalized?). Understanding whether the results of empirical studies or the anecdotes from previous projects are relevant to a new project is not easy. On the whole, "there is no simple answer regarding the use and performance of OO technologies" [6]. Nonetheless, developers should carefully examine the evidence regarding OOT to better understand potential benefits and risks for their specific project.

2.2.2 Project Characteristics

The second important question is:

(2) *What project characteristics are important with respect to OOT?*

Various attributes of a project might help determine whether OOT is an appropriate choice. Some of these attributes are measurable by conventional metrics specific to the software product; for example, the size, criticality, and complexity of the software. Other product-specific attributes include the maturity of the software requirements, and the applicability of OOT to the specific problem domain. Concerns were discussed regarding whether OOT is appropriate for *all* problem domains.

Other attributes of interest relate to the long-term plans for the product. Important considerations here include whether the software is a new product or part of a product family. This would impact upgrade and maintenance requirements. These factors are important when weighing the potential benefits of reuse that OOT might offer.

2.2.3 OOT Specific Resources

Another question that should be asked and answered is:

(3) *What project resources, specific to OOT, are needed?*

Once the project characteristics are known, it is important to evaluate resources specific for implementing OOT. Resources include those relevant to personnel who develop and approve the software product, and those relevant to managing processes and procedures for development and approval.

Personnel resources include OOT-specific training and experience, both at the individual level (such as the software developers and quality assurance personnel) and the corporate level. This includes training and experience with OO methods for modeling, design, analysis and testing, and with OO tools. Note that training and experience are concerns for regulators also, including Designated Engineering Representatives (DERs) within the company and certification authorities responsible for the software approval on the project.

Administrative resources include industry standards for OOT, such as those associated with the Object Management Group (OMG) standard for object-oriented modeling with the Unified Modeling Language (UML) [27] and standards for OO source code languages (for example, Ada95, Java, and C++). Other important standards include internal process standards that define life cycle activities and data associated with OOT and how those map to activities and data specified in DO-178B. Companies that commit to OOT may also have standards for packaging OO components for reuse. For example, standards may cover packaging development and verification artifacts from one project such that they do not conflict with other artifacts when they are brought together to build a larger or different system.

OO tools are another important resource to consider. Some OO tools introduce new levels of abstraction, such as the visual model level, that might not directly correspond to abstraction levels (high- or low-level requirements or design) in DO-178B. Factors to consider here include compatibility of new OO tools with existing tools and integrated development environments, notations, and processes; configuration management; and qualification costs.

The project characteristics together with the OOT specific resources within a company will influence the level of involvement, or degree of oversight, that the FAA has with a project. This is a non-trivial consideration with respect to both time and cost. The level of FAA involvement will dictate the number of software reviews, the stages of involvement, and the nature of the review [16]. This level of regulatory involvement is closely related to the fourth of the high-level questions raised at the workshop.

2.2.4 Regulatory Guidance

The fourth question is:

(4) *How should regulatory guidance, including DO-178B and the OOTiA handbook, be applied in a practical project?*

This question is really an abstraction of two more specific questions:

- Are all of the objectives in DO-178B compatible with OOT?
- How should the handbook be applied to a practical project, and is the handbook adequate?

As mentioned previously, the FAA is sponsoring the development of the OOTiA handbook to provide information specific to meeting the DO-178B objectives when using OOT. The handbook does not eliminate the need for compliance with DO-178B, but instead provides guidelines for how to use OOT to comply with the DO-178B objectives. Volume 3 of the handbook in particular is devoted to patterns and other information intended to facilitate this compliance. Although the handbook is not intended to become official FAA policy or guidance [28], the handbook will almost certainly influence the approval process for an OO program, and will likely influence future revisions of DO-178B.

Some participants in the brainstorming session argued that the existing guidance in DO-178B is sufficient to accommodate approval of an OO program. Some questioned the wisdom of generating an OOT-specific handbook, and wondered whether doing so implied a need for additional method-specific handbooks. For example, will we need handbooks for aspect- or goal-oriented programming? Other participants, including some regulators, however, argued in favor of the benefits that additional clarification and guidelines might provide to the industry in the short term (that is, until DO-178B is revised).

Ultimately, regulators and software developers should both understand the requirements that an OO system must satisfy for it to be approved, and how the system will be shown to satisfy these requirements [14]. Misunderstandings can result in substantial cost and schedule problems [30].

2.2.5 Technical Challenges

The final, and perhaps most difficult, question that should be asked and answered by anyone considering using OOT is:

(5) *What are the technical challenges in applying OOT to ensure the appropriate level of integrity required for the project?*

Specific questions raised in the session concerned how well the essential elements of software engineering can be done using OOT to ensure the appropriate level of integrity. Most of the questions within this grouping were about requirements, verification, or safety.

2.2.5.1 Requirements

Several questions asked whether OOT is an adequate approach for requirements development and implementation. That is, do OO approaches to requirements help with the correct specification and implementation of intended functionality? At least two points were raised: (1) the difference between approaches to requirements decomposition (functional versus object), and (2) documentation of requirements (text versus graphics).

With functional decomposition, the typical programming unit is some form of subprogram, such as a function, subroutine, or procedure. Each subprogram typically performs a single specific function, where good programming practice calls for maximizing functional cohesion within a subprogram and minimizing coupling between subprograms. Applications are built by sequencing these functional building blocks—"first do this, then do that." Verification, in turn, starts with the functionality of an individual subprogram and works its way up by testing increasing levels of functionality.

In contrast to functional decomposition, OOT focuses on objects and the operations performed by or to those objects. In an OO program, a class, which is a set of objects that share a common structure and a common behavior, is the structural element most comparable to a subprogram. Operations related to a given functional requirement often are distributed among objects associated with different classes. The concern here was whether the distribution of functionality inherent in OO systems complicates assurance of the intended functionality.

DO-178B organizes objectives for development and verification around the decomposition of requirements from high-level requirements to low-level requirements to source code. With a structured programming approach, the requirements specification is largely text-based, with diagrams such as Visio® diagrams, data and control flow diagrams, and sequence diagrams, included in the text. OOT is much more focused on a graphical representation of the system. Typically, requirements for OO systems are developed with use cases, scenarios, and various diagrams such as class, object, and activity diagrams. Determining how to map these, and their subsequent refinements, onto the DO-178B objectives was thought by session participants to be difficult. The number and diversity of the diagrams used to describe the system can add additional complication.

Requirements definition by any method is a significant challenge to developing a correct and safe system [18]. Developers should consider whether OOT makes this challenge easier or more difficult for their project.

2.2.5.2 Verification

In addition to the questions raised about the suitability of OOT for requirements development, a similar number of questions were raised about verification. The questions about verification are not unrelated to the concerns raised about requirements. The following sentiment exemplifies the opinion of many in the brainstorming session:

"object oriented programs are generally more complex than their procedural counterparts. This added complexity results from inheritance, polymorphism, and the complex data interactions tied to their use. Although these features provide power and flexibility, they increase complexity and require more testing" [1].

Several of the questions discussed in the session sought to explore the extent that OO software can be verified:

- Can we analyze OO software?
- Can we adequately test OO software?
- Can we determine the error cases unique to OOT?

That is, do we have the same level of confidence in our ability to adequately analyze and test OO programs as we do with structured programs? Some specific analysis issues included source to object code traceability, and control and data flow analysis. Several participants in the session argued for the application of static analysis and logic-based methods. Most of those participants would likely argue for static analysis and formal methods even in a functional approach. However, the broader question is whether additional verification methods are needed for OOT to meet the same level of assurance that could be obtained under a functional approach.

Lastly, many participants acknowledged the need for additional research to better understand error classes that are unique to OOT, such as research by Offutt [26], and to better understand the extent that existing methods are adequate for verifying OOT. Several error classes introduced by OOT have been submitted as entries to the issue list, and are discussed in section 3.

2.2.5.3 Safety

The final technical challenge mentioned in the questions concerns the ability to conduct effective system and software safety assessment. Participants discussed whether system and safety assessments can be easily and accurately derived from an OO program. Current safety analysis is often based on determining that a function, as implemented, is both correct and safe. In an OO program, operations related to a function can be widely distributed among objects that interact with each other by exchanging messages. Assessing the interaction among distributed objects complicates safety analysis and makes functionality difficult to trace. In [20], Nancy Leveson argues that engineers find that functional decomposition is a more natural approach to the design of control systems, and "That naturalness translates into easier to understand and review, easier to design without errors, easier to analyze to determine whether the system does what the engineer wants and does it safely."

Although safety analysis is not one of the life cycle activities specified in DO-178B, connections with safety assessment are mentioned in DO-178B [10] (e.g., DO-178B sections 5.1.2 j and 5.2.2d). Hence, the effect of OO design and implementation on safety analysis should be carefully considered.

The data from the Beyond the Handbook session represents only a small portion of the data collected in the OOTiA project. The majority of the data deals with issues specific to OO methods and languages; that is, the decision to use OOT is assumed. As might be guessed, many of the questions for deciding whether to use OOT are directly related to issues raised about OO features discussed in the next section.

2.3 Considerations After Making the Decision to Use OOT

In this section, key concerns about using OOT in aviation applications are distilled from the Issues and Comments about Object Oriented Technology in Aviation list on the OOTiA web site and from issues documented by certification authorities [7,22]. Throughout the discussion, entries from the issue list are cited and denoted by their tracking number on the web site (for example, IL 1). Mentioning an entry from the issue list does not imply its validity; some readers likely will dispute the validity of individual entries. The purpose of citing the entries is to give an account of the data that has been collected, and show the basis for the key concerns discussed in this section. The key concerns, not the individual issues, are what is important.

The issue list covers a wide range of topics. In this section, entries expressing similar concerns are grouped together, and the groups are sorted according to the DO-178B life cycle process that is most influenced[2] by that group. The software life cycle processes specified in DO-178B are:

- Planning Process
- Development Processes (requirements, design, code, and integration)
- Integral Processes (verification, configuration management, quality assurance, and certification liaison)

In addition to these life cycle processes, DO-178B has a section called *Additional Considerations* (section 12) that provides guidance for ancillary topics such as tools, previously developed software, and formal methods.

Each comment recorded on the issue list is mapped into one of the categories (Planning Process, Development Processes, Integral Processes, Additional Considerations). Comments related to the development processes are not separated with respect to requirements, design and such, because the distinction is not clear in many cases. The comments related to integral processes, however, are divided into concerns about verification and concerns about configuration management. No entries in the issue list dealt specifically with quality assurance or the certification liaison process. Finally, the additional considerations category deals almost exclusively with tools.

[2] Some concerns span multiple life cycle processes. Determining which process is most influenced is necessarily subjective. Overlap of concerns is evident throughout the discussion.

For each category, the key concerns relevant to DO-178B are briefly discussed. Appendix B provides a mapping of the entries in the issue list to these key concerns. Appendix C lists some of the issues raised in Webster's *Pitfalls of Object-Oriented Development* [Webster], which may also be important to consider.

2.3.1 Considerations for the Planning Process

Planning is typically the first consideration after the decision to use OOT has been made. The planning process in DO-178B specifies the development and integral process activities, environment, and standards for a project. Planning decisions about how OOT will be used to meet the DO-178B objectives are recorded in documents such as the Plan for Software Aspects of Certification, Software Development Plan, and integral process plans. The key concerns relevant to planning involve life cycle data, requirements, and standards.

2.3.1.1 Defining Life Cycle Data

A key concern pertains to defining how the life cycle data from an OO development process maps to the life cycle data specified in DO-178B section 11. OOT introduces new notations and models, such as behavior and implementation diagrams (use cases, class, sequence, component, deployment, activity, and statechart diagrams), that do not map directly with the data for requirements, design, and code in DO-178B. A description of the software life cycle data is typically included in the Plan for Software Aspects of Certification, as discussed in section 11.1e of DO-178B. Some specific questions from the issue list included:

- How does OO life cycle data map to the DO-178B section 11 life cycle data? e.g., What does "source code" mean in OO? What are requirements, design, and code in OOT? (IL 87)
- What are "low level requirements" for OO? (IL 77)

Another fundamental concern for planning relates specifically to requirements. Software requirements standards, as described in 11.6 of DO-178B, define the methods, rules, notations, and tools for developing high-level requirements. The concerns from the issue list covered two aspects of requirements: methods and notations.

2.3.1.2 Requirements Methods And Notations

OOT methods and notations specifically for requirements are a concern. The key concern is whether OO approaches to requirements definition (UML, for example) are adequate for all types of requirements. This includes: (1) concern about adequately capturing non-functional requirements (IL 75), and (2) concern about the tendency in OOT to group requirements in a graphical format—making identification of low-level requirements and derived requirements difficult, and complicating safety assessment (IL 79).

- Lower levels of decomposition may not be possible for some requirements (e.g., performance requirements). Levels of abstraction may be different than traditional. (IL 80)
- Philosophy of Functional Software Engineering - Most of the training, tools and principles associated with software engineering and assurance, including those of RTCA DO-178B, have been focused on a software function perspective, in that there is an emphasis on software requirements and design and verification of those requirements and the resulting design using reviews, analyses, and requirements-based (functional) testing, and RBT coverage and structural coverage analysis.
- Philosophy of Objects and Operations - Although generally loosely and inconsistently defined, OOT focuses on "objects" and the "operations" performed by and/or to those objects, and may have a philosophy and perspective that are not very conducive to providing equivalent levels of design assurance as the current "functional" approach. (IL 63)
- Addressing derived requirements for OO – how does this happen? How is it different than traditional and how does it tie up to the safety assessment? (IL 78)

2.3.1.3 *Restrictions*

The final topic relevant to planning deals with restrictions. Section 4.5c of DO-178B states, "the software development standards should disallow the use of constructs or methods that produce outputs that cannot be verified or that are not compatible with safety-related requirements" [10]. Some OO languages have features that could make it extremely difficult or impossible to satisfy the objectives of DO-178B. In many cases, a well-defined subset of the language may be identified and documented in the coding standards that will allow compliance to objectives for a given software level. For example, ANSI C++ has some language features, such as multiple inheritance, that may make it difficult to meet some DO-178B objectives. Two of the entries from the issue list spoke to the potential need for restrictions or special rules:

- Multiple inheritance should be avoided in safety critical, certified systems. (IL 38)
- How can we enforce the rules that restrict the use of specific OO features? (IL 58)

The key concern is that language features, such as multiple inheritance, should be evaluated carefully in the planning process and restrictions or rules established, documented, and followed as warranted for a particular project.

2.3.2 ***Considerations for Development Processes -- Requirements, Design, Code, and Integration***

Over 40 of the issues on the issue list are related to or affect development processes (requirements, design, code, and integration). Many of these comments describe ways that OO features promote complexity and ambiguity in the development products (requirements, design, and code), potentially making the two assurance principles (assurance of intended functionality and assurance of no unintended functionality) difficult to meet. Sections 6.3.1 and 6.3.2 of DO-178B describe objectives for ensuring that high and low-level requirements are accurate, unambiguous (written in terms that only allow a single interpretation), and consistent.

The following are the key concerns distilled from the issues in this category.

2.3.2.1 About Subtypes

Inheritance allows different objects to be treated in the same general way. Through inheritance, programmers can develop types by basing new type definitions on existing ones. In a type hierarchy, supertypes should capture the behavior that all of the descendent subtypes have in common. A clear understanding of how subtypes and supertypes are related is essential to having a correct type hierarchy. Two types of problems described in entries from the issue list are associated with subtyping.

2.3.2.1.1 Type Substitutability

The substitutability issue deals with the suitability of various subtypes that are possible at the point of a call. Whenever a system expects a value of type T, a value of type T', where T' is a subtype of T, can be substituted. The key concern here is that improper subtyping can result in unintended functionality that is difficult to detect. That is, how do we know, or provide assurance, that this substitution is always appropriate? (IL 42) The Liskov Substitution Principle (LSP) proposes a theoretical means to mitigate improper subtyping by constraining the behavior of subtypes [21]:

Subtype Requirement: Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

In theory, conformance to this principle would satisfy the assurance dilemma. However, in practice, this is difficult because the semantics of real programming languages differ considerably from the simple model used in [21]. For example, C++ and Java have semantics that are vastly more complex than the model of computation used in [21]. Showing that the semantics of a real programming language are equivalent to those used in [21] may be a difficult task. The following entries from the issue list speak to the concern:

- Use of inheritance (either single or multiple) raises issues of compatibility between classes and subclasses. (IL 17)

- A subclass either does not accept all messages that the superclass accepts or leaves the object in a state that is illegal in the superclass. This situation can occur in a hierarchy that should implement a subtype relationship that conforms to the Liskov substitution principle. (IL 22)
- A subclass computes values that are not consistent with the superclass invariant or superclass state invariants. (IL 23)
- Fundamental pre-requisite language issues need clarification prior to adopting LSP and Design by Contract (DBC). How can LSP be implemented using available languages? Strongly consider a language subset that is amenable to use of LSP and DBC. Concern is how far to take this subset. (IL 90)
- When a descendent adds an extension method that defines an inherited state variable, an inconsistent state definition can occur. (IL 95)

2.3.2.1.2 Inconsistent Type Use

Another subtyping problem is inconsistent type use. When a descendant class does not override any inherited method (that is, there is no polymorphic behavior), anomalous behavior can occur if the descendant class has extension methods resulting in an inconsistent inherited state. (IL 91)

2.3.2.2 About Subclasses

Just as with types, hierarchies of classes can be constructed, where subclasses are created from more abstract superclasses. A subclass automatically inherits all of the visible attributes and operations of the superclass; but can override inherited operations and add new attributes and operations. A number of different concerns were raised about subclasses.

2.3.2.2.1 Unclear Intent

Use of inheritance, polymorphism, and linkage can lead to ambiguity. (IL 10) With multiple inheritance, a subclass may have more than one superclass, so the same operation may be inherited from multiple sources. Consequently, the intent of the operation at the subclass level might not be clear. Cuthill notes that overuse of inheritance, particularly multiple inheritance, can lead to unintended connections among classes [9]. The key concern is that the original intent for a subclass or operation may not be clear. Examples of concerns about unclear intent from the issue list include:

- Multiple interface inheritance can introduce cases in which the developer's intent is ambiguous. (when the same definition is inherited from more than one source is it intended to represent the same operation or a different one?) (IL 15)
- When the same operation is inherited by an interface via more than one path through the interface hierarchy (repeated inheritance), it may be unclear whether this should result in a single operation in the subinterface, or in multiple operations. (IL 27)
- A subclass may be incorrectly located in a hierarchy because the complete definition/intention of a class may not be clear. (IL 21)
- When a subinterface inherits different definitions of the same operation [aa result of redefinition along separate paths]it may be unclear whether/how they should be combined in the resulting subinterface. (IL 28)
- Multiple inheritance complicates the class hierarchy. (IL 33)
- Top-heavy multiple inheritance and very deep hierarchies (six or more subclasses) are error-prone, even when they conform to good design practice. The wrong variable type, variable, or method may be inherited, for example, due to confusion about a multiple inheritance structure. (IL 24)
- Overuse of inheritance, particularly multiple inheritance, can lead to unintended connections among classes, which could lead to difficulty in meeting the DO-178B/ED-12B objective of data and control coupling. (IL 25) (IL 37)

- Polymorphic, dynamically bound messages can result in code that is error prone and hard to understand. (IL 7)
- Use of multiple inheritance can lead to “name clashes” when more than one parent *independently* defines an operation with the same signature. (IL 29)
- When *different* parent interfaces define operations with different names but compatible specifications, it is unclear whether it should be possible to merge them in a subinterface. (IL 30)

2.3.2.2.2 Overriding

Overriding is the redefinition of an operation or method in a subclass. The key concern here is that unintentionally overriding an operation is easy in some OO languages because of the lack of restrictions on name overloading (the use of the same name for different operators or behavioral features, operations or methods, visible within the same scope). The consequence is that a method of the expected name but of a different type might be called in a program.

- It is important that the overriding of one operation by another and the joining of operations inherited from different sources always be intentional rather than accidental. (IL 32)
- A subclass-specific implementation of a superclass method is [accidentally] omitted. As a result, that superclass method might be incorrectly bound to a subclass object, and a state could result that was valid for the superclass but invalid for the subclass owing to a stronger subclass invariant. For example, object-level methods like `isEqual` or copy are not overridden with a necessary subclass implementation. (IL 20)
- It is unclear whether the normal overload resolution rules should apply between operations inherited from different superinterfaces or whether they should not (as in C++). (IL 31)

Offutt has identified the following five classes of errors associated with overriding [26]:

1 If a computation performed by an overriding method is not semantically equivalent to the computation of the overridden method with respect to a variable, a behavior anomaly can result. (IL 94) This is referred to as a State Defined Incorrectly (SDI) problem.

2 If a descendant class provides an overriding definition of a method which uses variables defined in the descendant’s state space, a data flow anomaly can occur. (IL 96) This is referred to as an Anomalous Construction Behavior (ACB1) problem.

3 If a descendant class provides an overriding definition of a method which uses variables defined in the ancestor’s state space, a data flow anomaly can occur. (IL 97) This is referred to as an Anomalous Construction Behavior (ACB2) problem.

4 If refining methods do not provide definitions for inherited state variables that are consistent with definitions in an overridden method, a data flow anomaly can occur. (IL 92) This is referred to as a State Definition Anomaly (SDA) problem.

5 When private state variables exist, if every overriding method in a descendant class doesn’t call the overridden method in the ancestor class, a data flow anomaly can exist. (IL 99) This is referred to as a State Visibility Anomaly (SVA) problem.

As mentioned above, overriding is affected by the use of overloading. In theory, overloading enhances readability when the overloaded operators, operations or methods are semantically consistent. (IL 60) However, overloaded operators, operations, and methods contribute to confusion and human error when they introduce methods that have the same name but different semantics.

2.3.2.3 About Memory Management and Initialization

According to DO-178B (section 11.10), the software design data should discuss limitations for memory and the strategy for managing memory and its limitations. Memory management involves making accessible the memory needed for a program's objects and data structures from the limited resources available, and recycling that memory for reuse when it is no longer required. “The basic problem in managing memory is knowing when to keep the data

it contains, and when to throw it away so that the memory can be reused. This sounds easy, but is, in fact, such a hard problem that it is an entire field of study in its own right" [23].

2.3.2.3.1 Indeterminate Execution Profiles

One problem with memory allocators is that regardless of the allocation algorithm used, allocating and deallocating memory repeatedly leads to fragmentation. Some algorithms, of course, lead to more fragmentation than others. In any case, periodic reorganization of memory is needed to reduce fragmentation. The key concern is that many traditional allocation and deallocation algorithms are unpredictable in terms of their worst-case memory use and execution times, resulting in indeterminate execution profiles (IL 66).

2.3.2.3.2 Initialization

Incorrect initialization of variables and constants is dealt with in sections 6.3.4 and 6.4.3 of DO-178B. In OO programs, class hierarchies (deep hierarchies in particular) may lead to initialization problems. The key concern is that a subclass method might be called (via dynamic dispatch) by a higher level constructor before the attributes associated with the subclass have been initialized. (IL 19) This can lead to the incomplete (failed) construction problem identified by Offutt [26]. (IL 98) According to Offutt, there are two possible faults, depending on programming language:

"First, the construction process may have assigned an initial value to a particular state variable, but it is the wrong value. That is, the computation used to determine the initial value is in error. Second, the initialization of a particular state variable may have been overlooked. In this case, there is a data flow anomaly between the constructor and each of the methods that will first use the variable after construction (and any other uses until a definition occurs)" [26].

2.3.2.4 About Dead or Deactivated Code

Reuse is an important design goal for many OO programs. However, the requirements, design, and code of a reusable component might cover more functionality than required by the system being certified, which raises concerns about dead and deactivated code. The glossary of DO-178B describes dead and deactivated code as follows:

"Dead code - Executable object code (or data) which, as a result of a design error cannot be executed (code) or used (data) in a operational configuration of the target computer environment and is not traceable to a system or software requirement. An exception is embedded identifiers" [10].

"Deactivated code - Executable object code (or data) which by design is either (a) not intended to be executed (code) or used (data), for example, a part of a previously developed software component, or (b) is only executed (code) or used (data) in certain configurations of the target computer environment, for example, code that is enabled by a hardware pin selection or software programmed options" [10].

Section 6.4.4.3 of DO-178B requires that dead code be removed and analysis performed to assess the effect and need for reverification, and Section 5.4.3 of DO-178B requires deactivated code to be verified (analysis and test) to prove that it cannot be inadvertently activated. Hence, there is a trade-off between the benefit of reuse and the cost of additional verification of deactivated code or removal of dead code.

2.3.2.4.1 Identifying Dad and Dectivated Code

A specific concern raised in the issue list is the difficulty of identifying dead code and deactivated code in OO programs. Dead or deactivated code can result when (a) methods of a class are not called in a particular application; (b) methods of a class are overridden in all subclasses; or (c) attributes of a class are not accessed in a particular application [8]. For example, no instances of a superclass might be used if all of the subclasses override a particular method. The following entries from the issue list deal with dead/deactivated code in general:

- The difference between dead and deactivated code is not always clear when using OOT. Without good traceability, identifying dead versus deactivated code may be difficult or impossible. (IL 70)
- When a design contains abstract base classes, portions of the implementations of these classes may be overridden in more specialized subclasses, resulting deactivated code. (IL 71)

- Reusability is one of the objectives of OO development, but reusable components may be hard to trace because they are designed to support multiple usages of the same component. Reusable components may also have functionality that may not be used in every application. (IL 106)

2.3.2.4.2 Libraries and Frameworks

Dependence on libraries is a concern for safety-critical systems because it is often unclear what is happening in the object libraries. Libraries may not have been developed with safety-critical applications in mind and may not have the integrity required for such applications [7]. The key concern here is that dead or deactivated code can result from using general-purpose libraries and OO frameworks when all elements of the libraries or frameworks are not used. This key concern applies equally well to non-OO systems. However, some might argue that dependence on libraries and frameworks may be more extensive in an OO system. In any case, use of libraries must be carefully considered and verified for proper functionality.

- Deactivated Code will be found in any application that uses general purposed libraries or object-oriented frameworks. (Note that this is the case where unused code is NOT removed by smart linkers.) (IL 1)
- How can deactivated code be removed from an application when general purpose libraries and object-oriented frameworks are used but not all of the methods and attributes of the classes are needed by a particular application? (IL 57)

2.3.3 Considerations for Integral Processes

Complexity and ambiguity that are concerns for the development processes are concerns also for the integral processes. The entries from the issue list dealing with integral processes are partitioned into three categories: verification, configuration management, and traceability. Although traceability is not called out as an integral process in DO-178B, it is categorized here under integral processes because traceability data is often collected in conjunction with verification activities.

The key concerns within each of the three categories are discussed below. For each of the key areas, we try to differentiate between the effect of the OO techniques and OO language constructs.

2.3.3.1 About Verification - Analysis

Verification, in the DO-178B context, examines the relationship between the software product and the requirements. The goal is to use reviews, analyses, and tests to detect and report errors introduced during the development processes. OO techniques, especially dynamic dispatch and polymorphism, can complicate various verification activities required by DO-178B (IL 9). Program control flow can be difficult to predict (if it can be predicted at all) because polymorphism forces method binding to be delayed until execution time. Execution-time circumstances can cause a single line of source code to mean many different things depending upon specific data values that the program sees. For example, given a function *f*(*x*), which *f*() to call depends on which class *x* belongs to, which might be multiple classes depending on the run-time state of the system. This is not a problem of doubt about a conditional statement—it is doubt about what a specific function call means because of dynamic dispatch [17]. This particular problem complicates testing and many different types of analyses including flow and coupling analysis, structural coverage analysis, and source to object traceability as discussed below.

2.3.3.1.1 Flow Analysis

Data and control flow analysis must be performed for software levels A-C to confirm data and control coupling between code components. OO features tend to make data and control coupling relationships more complicated and obscure than in software developed using procedural languages. Dynamic dispatch complicates flow analysis, as described above, because it might be unclear which method in the inheritance hierarchy is going to be called. OO design, in general, encourages the development of many small, simple methods to perform the services provided by a class. Determining the correctness of control flow decisions requires analysis of how individual data objects that control the execution flow of the software are created and maintained: where and when are the objects created or destroyed, when and when are the values of the objects set, and how are any potential “shared data” conflicts controlled. The key concern in OO programs is that decision points use data objects with values that are maintained

in other parts of the software that might be remote from the proximate path of execution—making flow analysis difficult [33].

OOT also encourages hiding the details of the data representation (that is, attributes) behind an abstract class interface. Suggested best practice is that attributes of an object should be private, and access to them only provided through the methods appropriate to the class of the object. Being able to access attributes only through methods makes the interaction between two or more objects implicit in the code, complicating analysis.

Many of the relevant entries from the issue list express questions about how to do the analysis.

- How can we meet the control and data flow analysis requirements of DO-178B with respect to dynamic dispatch? (IL 56)
- Flow analysis, recommended for Levels A-C, is complicated by dynamic dispatch (just which method in the inheritance hierarchy is going to be called?). (IL 2)
- Flow analysis and structural coverage analysis, recommended for Levels A-C, are complicated by multiple implementation inheritance (just which of the inherited implementations of a method is going to be called and which of the inherited implementations of an attribute is going to be referenced?). The situation is complicated by the fact that inherited elements may reference one another and interact in subtle ways which directly affect the behavior of the resulting system. (IL 16)

OO language features such as inlining can also complicate flow analysis because inlining can cause substantial differences between the flow apparent in the source code and the actual flow in the object code.

- Flow Analysis, recommended for levels A-C, is impacted by Inlining (just what are the data coupling and control coupling relationships in the executable code?). The data coupling and control coupling relationships can transfer from the inlined component to the inlining component. (IL 43)

2.3.3.1.2 Structural Coverage Analysis

Structural coverage analysis is required in DO-178B for software levels A-C. The intent of structural coverage analysis, in the DO-178B context, is to complement requirements-based testing by: (1) providing evidence that the code structure was verified to the degree required for the applicable software level; (2) providing a means to support demonstration of absence of unintended functions; and, (3) establishing the thoroughness of requirements-based testing [11].

Structural coverage analysis is complicated by dynamic dispatch because structural coverage changes when going from subclass to superclass. The key concern is that structural coverage in an OO program is not meaningful unless coverage measurements are context dependent; that is, based on the class of the specific object on which the methods were executed. “Coverage achieved in the context of one derived class should not be taken as evidence that the method has been fully tested in the context of another derived class” [15]. The following entries in the issue list attest to the complications:

- Structural coverage analysis, recommended for Levels A-C, is complicated by dynamic dispatch (just which method in the inheritance hierarchy does the execution apply to?). (IL 5)
- The use of inheritance and polymorphism may cause difficulties in obtaining structural coverage, particularly decision coverage and MC/DC (IL 11)
- The unrestricted use of certain object-oriented features may impact our ability to meet the structural coverage criteria of DO-178B. (IL 48)
- Statement coverage when polymorphism, encapsulation or inheritance is used. (IL 49)
- How can we meet the structural coverage requirements of DO-178B with respect to dynamic dispatch? There is cause for concern because many current Structural Coverage Analysis tools do not “understand” dynamic dispatch, i.e. do not treat it as equivalent to a call to a dispatch routine containing a case statement that selects between alternative methods based on the run-time type of the object. (IL 55)

The following entries from the issue list refer to the effect that OO language constructs such as inlining and templates have on structural coverage analysis:

- With inlining, the “logical” coverage of the inline expansions on the original source code is not clear. This is generally only a problem when inlined code is optimized. If statements are removed from the inlined version of a component, then coverage of the inlined component is no longer sufficient to assert coverage of the original source code. (IL 45)
- Inlining may affect tool usage and make structural coverage more difficult for levels A, B, and C. (IL 47)
- Templates can be compiled using code sharing or macro-expansion. Code sharing is highly parametric, with small changes in actual parameters resulting in dramatic differences in performance. Code coverage, therefore, is difficult and mappings from a generic unit to object code can be complex when the compiler uses the "code sharing" approach. (IL 52)

Code sharing involves the sharing of code by more than one class or component, for example, by means of implementation inheritance or delegation. There are many ways to support code sharing. One risk is that inheritance can be misused to support only the sharing of code and data structure, without attempting to follow behavioral subtyping rules.

2.3.3.1.3 Timing and Stack Analysis

Timing analysis, worst-case execution time in particular, and stack usage are both part of review and analysis of source code in DO-178B section 6.3.4f. Stack overflow errors are listed in section 6.4.3f of DO-178B as errors that are typically found in requirements-based hardware/software integration testing. Timing and stack analysis are complicated by certain implementations of dynamic dispatch. With some implementations of dynamic dispatch, it is difficult to know just how much time will be expended determining which method to call. (IL 3) If polymorphism and dynamic binding are implemented, stack size can grow, making analysis of the optimal stack size difficult. (IL 107)

Timing and stack analysis are also affected by inlining, templates, and macro-expansion. Inline expansion can eliminate parameter passing, which can affect the amount of information pushed on the stack as well as the total amount of code generated. This, in turn, can affect the stack usage and timing analysis.

- Stack Usage and Timing Analysis, recommended for levels A-D, are impacted by Inlining (just what are the stack usage and worst-case timing relationships in the executable code?). Since inline expansion can eliminate parameter passing, this can affect the amount of information pushed on the stack as well as the total amount of code generated. This, in turn, can affect the stack usage and the timing analysis. (IL 44)
- Templates are instantiated by substituting a specific type argument for each formal type parameter defined in the template class or operation. Passing a test suite for some but not all instantiations cannot guarantee that an untested instantiation is bug free. (IL 50)
- Macro-expansion can result in memory and timing issues, similar to those identified for inlining. (IL 53)

2.3.3.1.4 Source to Object Trace

Source to object code traceability tends to be a controversial issue; object orientation does not improve the situation. As discussed in DO-178B, for level A software, it is necessary to establish whether the object code is directly traceable to the source code. If the object code is not directly traceable to the source code, then additional verification should be performed [10]. Dynamic dispatch complicates source to object code traceability because it might be difficult to determine how the dynamically dispatched call is represented in the object code. (IL 6) In addition, source to object code correspondence will vary between compilers for inheritance and polymorphism, along with constructors/destructors and other language helper functions. (IL 12) Additional entries from the issue list related to source to object traceability include:

- Dynamic dispatch presents a problem with regard to the traceability of source code to object code that requires “additional verification” for level A systems as dictated by DO-178B section 6.4.4.2b. (IL 8)
- Are there unique challenges for source to object code traceability in non-Level A systems? Where should this be addressed? Multiple tools and ways of addressing source to object traceability? (IL 81)

Some OO language features, such as inlining and implicit type conversion, can also complicate source to object code traceability:

- Conformance to the guidelines in DO-178B concerning traceability from source code to object code for Level A software is complicated by inlining (is the object code traceable to the source code at all points of inlining/expansion?). Inline expansion may not be handled identically at different points of expansion. This can be especially true when inlined code is optimized. (IL 46)
- Implicit type conversion raises certification issues related to source to object code traceability, the potential loss of data or precision, and the ability to perform various forms of analysis called for by DO-178B including structural coverage analysis and data and control flow analysis. It may also introduce significant hidden overheads that affect the performance and timing of the application (IL 59)

2.3.3.2 About Verification - Testing

Testing in DO-178B has two high-level objectives tied to the two fundamental assurance principles: demonstrating that the software satisfies its requirements, and demonstrating (with a high degree of confidence) that errors that could lead to unacceptable failure conditions have been removed (section 6.4 of DO-178B). As mentioned under the verification concerns from the Beyond the Handbook session, OO features such as inheritance and polymorphism increase the complexity of OO programs and require more testing. The topic of testing OO programs is huge in scope with many books[3] and considerable research devoted to the subject.

Issue list entries identify two general challenges with respect to testing: requirements testing and test case reuse.

2.3.3.2.1 Requirements Testing

The first challenge goes back to the difference between the functional and object perspective. Three levels of tests are called out in DO-178B testing activities: low-level tests, software integration tests, and hardware/software integration tests. The key concern for requirements testing is that the mapping of function-oriented test cases to an object-oriented implementation might not be obvious since the basic unit of testing in an OO program is not a function or a subroutine, but an object or a class. Also, test coverage of high-level and low-level requirements will likely require different testing strategies and tactics from the traditional structured approach because information hiding and abstraction techniques decrease or complicate the observability of low-level functions.

- How do you determine functional coverage of the low level requirements? (IL 62)
- Software integration testing may be [improperly] avoided because the high level of interaction between a great number of objects could lead to an excessive number of test cases. (IL 64)

2.3.3.2.2 Test Case Reuse

The second challenge deals with the reuse of test cases or, more specifically, determining the appropriate reuse of test cases. Requirements-based testing is complicated by inheritance, dynamic dispatch, and overriding because it might be difficult to determine how much testing at a superclass level can be reused for its subclasses. (IL 4)

- Inheritance and overriding raise a number of issues with respect to testing: “Should you retest inherited methods? Can you reuse superclass tests for inherited and overridden methods? To what extent should you exercise interaction among methods of all superclasses and of the subclass under test?”(IL 18)

2.3.3.3 About Configuration Management

Although configuration management is addressed in only a few entries from the issue list, the comments are worth noting here because configuration management is an essential element of achieving regulatory approval. In DO-178B, configuration identification and control involve defining what constitutes a configuration item, and defining processes for controlling (baselining and changing) those items.

[3] Some of the books themselves are huge; for example Binder’s *Testing Object-Oriented Systems, Models, Patterns, and Tools* [Binder] is over 1000 pages!

2.3.3.3.1 Configuration Identification

Concerns are raised in entries from the issue list regarding what constitutes a configuration item. Section 7.2.1 of DO-178B discusses the need for unambiguous labeling of each configuration item so that a basis is established for control. The key concern involves the uniqueness of configuration items.

- Configuration management may be difficult in OO systems, causing traceability problems. If the objects and classes are considered configuration items, they can be difficult to trace, when used multiple times in slightly different manners. (IL 76)

2.3.3.3.2 Configuration Control

The key concern for configuration control is how OO tools and modeling languages such as UML might affect the way configuration items are managed and changed.

- Configuration management and incremental development of OO projects and tools. When configuration management comes into play during the development process may be different than our current practices, when using an UML tool. (IL 88)
- Change impact analysis may be difficult or impossible due to difficulty in tracing functional requirements through implementation. (IL 74)
- Multiple inheritance complicates configuration control. (IL 34)

2.3.3.4 About Traceability

Although traceability is not called out as a life cycle process in DO-178B, traceability plays a large role in providing assurance of intended functionality and assurance of the absence of unintended functionality. For that reason, traceability is included in considerations for integral processes. DO-178B guidelines require traceability between (a) system requirements and software requirements to enable verification of the complete implementation of the system requirements and give visibility to derived requirements; (b) low-level and high level requirements to verify the architectural design decisions, give visibility to derived requirements, and demonstrate complete implementation of high-level requirements; and (c) source code and low-level requirements to enable verification of the absence of undocumented source code and verification of the complete implementation of the low-level requirements (section 5.5 of DO-178B).

2.3.3.4.1 Function versus Object Tracing

The key concern is that traceability of functional requirements through implementation might be lost or difficult with an OO program because of mismatches between function-oriented requirements and an object-oriented implementation (IL 61). Providing traceability from a code sequence to a specific requirement might be difficult because operations related to a function might be widely distributed among objects. Inheritance, polymorphism, and overloading exacerbate the problem by increasing the complexity of interaction among distributed objects.

- The use of OO methods typically leads to the creation of many small methods which are physically distributed over a large number of classes. This, and the use of dynamic dispatch, can make it difficult for developers to trace critical paths through the application during design and coding reviews. (IL 69)

2.3.3.4.2 Complexity

Another key concern for traceability is that class hierarchies, especially those constructed through multiple inheritance, can become overly complex, making traceability difficult.

- Polymorphic and overloaded functions may make tracing and verifying the code difficult. (IL 13)
- When inheritance is used in the design, special care must be taken to maintain traceability. This is particularly a concern if multiple inheritance is used. (IL 35)

- Class hierarchies can become overly complex, which complicates traceability. Generalization, weak aggregation, strong aggregation, association and composition are some of the relations that can be used to create the class diagrams. (IL 104)

2.3.3.4.3 Tracing through OO Views

OO requirements, design, and implementation might have multiple "views"; for example, the class diagram in the logical view. A key concern is that behavior of the classes and how they interact together to provide the required function might not be visible in any single view. Dynamic information concerning control flow or data flow might not be visible [29].Unfortunately, many current OO tools do not currently provide a mechanism to trace requirements through multiple views and notations. IL 72 notes that traceability is made more difficult because there is often a lack of OO methods or tools for the full software lifecycle.

2.3.3.4.4 Iterative Development

Some OO programs are developed through iteration. The key concern here is that the risk of losing traceability may increase when using an iterative development process because of increased changes to development artifacts. Although this is certainly a traceability issue, this could also be considered when planning for the development process.

- Iterative development is often desired in OO implementation. Each iterative cycle has its own requirements (normally a set of Use Cases), design, implementation, and test. There is a risk of losing traceability when using iterative development.This can be caused by adding or changing requirements, design, or implementations. (IL 105)

2.3.4 *Additional Considerations*

Ten of the 103 issues on the issue list were categorized under Additional Considerations, most of these being tool issues. DO-178B is concerned with identifying the tools that are used through the software life cycle environment planning process, controlling those tools, and qualifying them. A wide range of OO tools exist to support development and verification. Some OO tools provide support for developing design and code through framework libraries of patterns, templates, generics, and classes, and also a framework to automatically generate source code from models. Three primary areas of concern regarding tools are: (1) capability to meet DO-178B objectives, (2) long-term maintainability and maturity of tools and tool environments and (3) qualification.

2.3.4.1 Tool Capability

A key concern about OO tools is whether the introduction of these tools in the development process contributes to the integrity of the software products or adds additional burden for verification.

- Current visual modeling tools that are used for OO development make use of frameworks for automatic code generation, replacing tedious programming tasks. Frameworks may include patterns, templates, generics, and classes in ways requiring new verification approaches. The tool's framework may or may not enforce requirements, design and coding standards. (IL 101)
- Current visual modeling tools that are used for OO development provide a capability to generate source code directly from UML models. Most of the existing UML tools today can use visual modeling diagrams to construct models and generate source code from these models. The level of source code generation depends on the tool and on the user of the tool. It is unclear how such tools may be used in aviation projects. (IL 102)
- The current structural coverage tools available may not "be aware" or have visibility to the internals of inherited methods and attributes and polymorphic references supported with dynamic binding such that they can provide a reliable measurement of the structural coverage achieved by the requirements-based testing. (IL 103)

2.3.4.2 Tool Environments

DO-178B provides guidelines on software life cycle environment control in section 7.2.9 to ensure that tools that are used are properly identified, controlled, and retrievable. An important concern about OO tools is that the rapid rate of evolution of these tools may conflict with the needs of aviation software developers to support tools for relatively long periods of time. Concern was also expressed about anticipating new tool types. Specific entries in the issue list pertaining to tool environments include:

- Maturity/long-term support of tools. Tool manufacturers may not realize the long-life need of tools. Is this a higher risk in the OO environment? Education for both the tool and aviation communities to understand the specific needs for tool manufacturers and aircraft manufacturers. Not necessarily OO-specific, but might be more prevalent with OO. (IL 85)
- Maintaining tool environment, archives, … when licenses are involved is not clear. May need to have some kind of "permanent license" to support safety and continued airworthiness of the aircraft. OO more dependent on tools, but not necessarily an OO-specific issue. (IL 84)
- Are there other types of OO tools that need to be addressed? Need to anticipate other classes of tools that may come onto the scene; e.g., traceability tool for OO, transformation tools, CM tools, refactoring tools (tool to restructure source code to meet new requirements). (IL 86)

2.3.4.3 Tool Qualification

According to section 12.2 of DO-178B, tool qualification is needed when processes in DO-178B are eliminated, reduced, or automated by the use of a software tool without its output being verified. Qualification requirements differ depending on whether a tool is a development tool (whose output is part of the software) or a verification tool (that cannot introduce errors, but might fail to detect them). A concern raised in the entries from the issue list is whether new or different qualification criteria are needed for OO specific tools such as visual modeling tools or OO frameworks that serve as the basis for multiple data items. The following entries from the issue list deal with these concerns:

- Is there another "class" of tool qualification for visual modeling tools to demonstrate the integrity of these tools? Not necessarily automating a step, but are looking to make sure the tool is doing what you want. How to ensure consistency of the tools (validating the tool)? How to validate the tool when changes occur? (IL 82)
- Auto-test and code generation tools – what are the concerns when a single tool generates code and test from the same model? The concern is with the independence – same input and same tool. (IL 83)
- When using OO tools to develop software requirements, design and implementation, it is beneficial to work at the visual model level, especially when using UML. When working with OO tools, configuration management might be done at the modeling level (i.e., diagrams). This may cause a concern when the OO tools can introduce subtle errors into the diagrams. (IL 100)

2.4 Open Issues

During OOTiA Workshop 2, participants were invited to raise and discuss any issue that they thought had not been adequately covered so far in the draft OOTiA handbook. The comments spanned a wide range of topics—some technical (such as garbage collection and exception handling), and some organizational (such as developer and auditor capability and future use of formal methods). As noted previously, the listing of a comment from the brainstorming session does not imply the validity of that particular comment. The comments are reported here in the spirit of accounting for the data collected through the OOTiA activities. The OOTiA committee has not attempted to deal with these comments.

The list below is an unedited listing of the comments.

- Garbage Collection and memory management. Are we "going to fly" garbage collection? There has been a lot of work done in this area and nearing maturity. Destructors/finalizers are a partial alternative, but raise issues of predictability and performance.

- Exception handling. Use of this feature raises issues with control flow, run-time support, real-time predictability, and deactivated code.
- Concurrency. Use of this feature raises issues with control flow, run-time support, real-time predictability, and deactivated code.
- What characteristics of a program make it OO? What tells you that you are dealing with an OO program? (dynamic dispatching)
- Distinction between object-based and object oriented program.
- What level of capability should we expect from developers and auditors – and how would be measure that?
- Does OOT introduce a level of complexity that is not understandable by humans? (spaghetti data structures) Does OOT allow you to more easily slip down the path of complexity?
- Does doing OO change the picture that already exists?
- Should there be consideration of maintenance of large OO programs? Should the handbook offer guidance for long term maintainability?
- Moving target of language standard (e.g., Java moves every 6 months)
- Should we be addressing other programming paradigms?
- What about the consistency of guidelines among ground-based, space-based, and airborne systems?
- Is it worth looking for other instances of object oriented use that should be advised against (such as those given in the multiple inheritance chapter)?
- Mapping OO life cycle data to DO-178B section 11 life cycle data; e.g., what are requirements, design, and code in OO?
- How do you review code that has been generated by a non-qualified code generator?
- If you are going to go OO, you may require more processing power and memory for the delivered system than if you had not used OO.
- Formal Methods:
    - Why aren't correct-by-construction and static verification recognized as valuable within the aviation community?
    - Ignorance about static verification.
    - Documentation of best practices that include formal methods for producing better software.
    - Formal methods should be included in DO-178C, acknowledging the maturity of formal methods.
    - Determine the gain you get from formal methods by showing how it affects Annex A.

2.5 Summary

Identifying safety and certification concerns is an important step in developing guidelines for safely using OOT in aviation applications. Because OOT has been around for a while, there is industrial experience to help shed light on possible problems. In this report, we focus specifically on potential pitfalls to using OOT in aviation applications, as reported in the form of concerns and questions about OOT that have been collected through the OOTiA project web site and workshops. In general, two sets of challenges are presented: challenges to consider before making the decision to use OOT on a program, and challenges to consider once that decision is made.

During a brainstorming session at the second OOTiA workshop, participants proposed that the following subjects should be evaluated as part of the decision-making process for using OOT:

- Reality of the benefits of using OOT

- Project characteristics
- Resources specific to implementing OOT
- Regulatory guidance
- Technical challenges in the areas of requirements, verification, and safety.

Other challenges to safely implementing OOT in compliance with DO-178B were captured through the OOTiA web site on a list of Issues and Comments about Object Oriented Technology in Aviation. The key areas of concern, as outlined below, are organized with respect to DO-178B life cycle activities:

2.3.1 Considerations for the Planning Process
- 2.3.1.1 Defining Life Cycle Data
- 2.3.1.2 Requirements Methods and Notations
- 2.3.1.3 Restrictions

2.3.2 Considerations for Development Processes -- Requirements, Design, Code, and Integration
- 2.3.2.1 About Subtypes
  - 2.3.2.1.1 Type substitutability
  - 2.3.2.1.2 Inconsistent Type Use
- 2.3.2.2 About Subclasses
  - 2.3.2.2.1 Unclear Intent
  - 2.3.2.2.2 Overriding
- 2.3.2.3 About Memory Management and Initialization
  - 2.3.2.3.1 Indeterminate Execution Profiles
  - 2.3.2.3.2 Initialization
- 2.3.2.4 About Dead or Deactivated Code
  - 2.3.2.4.1 Identifying Dead and Deactivated Code
  - 2.3.2.4.2 Libraries and Frameworks

2.3.3 Considerations for Integral Processes
- 2.3.3.1 About Verification - Analysis
  - 2.3.3.1.1 Flow Analysis
  - 2.3.3.1.2 Structural Coverage Analysis
  - 2.3.3.1.3 Timing Analysis
  - 2.3.3.1.4 Source to Object Trace
- 2.3.3.2 About Verification - Testing
  - 2.3.3.2.1 Requirements Testing
  - 2.3.3.2.2 Test Case Reuse
- 2.3.3.3 About Configuration Management
  - 2.3.3.3.1 Configuration Identification
  - 2.3.3.3.2 Configuration Control
- 2.3.3.4 About Traceability
  - 2.3.3.4.1 Function Versus Object Tracing
  - 2.3.3.4.2 Complexity
  - 2.3.3.4.3 Tracing through OO Views
  - 2.3.3.4.4 Iterative Development

2.3.4 Additional Considerations
- 2.3.4.1 Tool Capability
- 2.3.4.2 Tool Environments
- 2.3.4.3 Tool Qualification

For each of the above topics, key concerns are identified based on input to the OOTiA program. This list is certainly incomplete; however it provides a starting point for developing guidelines for the safe use of OOT in aviation applications. Volume 3 of the OOTiA handbook provides guidelines for developing OOT applications in systems to be certified by the FAA. Volume 4 provides an approach for certification authorities and designees to ensure that OOT issues have been addressed in the projects they are reviewing or approving.

2.6 References

1. Alexander, Roger T., September/ October 2001, "Improving the Quality of Object-Oriented Programs," *IEEE Software*, pp. 90-91.

2. Aerospace Vehicle Systems Institute, Guide to the Certification of Systems with Embedded Object-Oriented Software, version 1.2, 31 October 2001

3. Aerospace Vehicle Systems Institute, Guide to the Certification of Systems with Embedded Object-Oriented Software, version 1.6.

4. Basili, V., L. Briand and W. Melo, "How Reuse Influences Productivity in Object-Oriented Systems," *Communications of the ACM*, vol. 39, no. 10, 1996, pp. 104-116.

5. Binder, Robert V., Testing Object-Oriented Systems, Addison-Wesley, Reading, MA, 2000.

6. Briand, L., E. Arisholm, S. Counsell, F. Houdek, and P. Thévenod-Fosse, "Empirical Studies of Object-Oriented Artifacts, Methods, and Processes: State of the Art and Future Directions," Technical Report ISERN-99-12, 1999.

7. Certification Authorities Software Team (CAST), "Object-Oriented Technology (OOT) in Civil Aviation Projects: Certification Concerns," Position Paper CAST-4, January 2000, available at http://www2.faa.gov/certification/aircraft/av-info/software/CAST_Papers.htm. Visited on 29 July 2003.

8. Certification Authorities Software Team (CAST), "Use of the C++ Programming Language," Position Paper CAST-8, January 2002, available at http://www2.faa.gov/certification/aircraft/av-info/software/CAST_Papers.htm. Visited on 29 July 2003.

9. Cuthill, Barbara, "Applicability of Object-Oriented Design Methods and C++ to Safety-Critical Systems", *Proceedings of the Digital Systems Reliability and Safety Workshop*, 1993.

10. RTCA, Inc., Software Considerations in Airborne Systems and Equipment Certification, RTCA/DO-178B, December 1992, Washington, D. C.

11. RTCA, Inc., Final Report for Clarification of DO-178B "Software Considerations in Airborne Systems and Equipment Certification", RTCA/DO-248B, 12 October 2001, Washington, D. C.

12. RTCA, Inc., Guidelines for Communication, Navigation, Surveillance, and Air Traffic Management (CNS/ATM) Systems Software Integrity Assurance, RTCA/DO-278, 5 March 2002, Washington, D. C.

13. Glass, Robert L, May/June 2002, "The Naturalness of Object Orientation: Beating a Dead Horse?" *IEEE Software*, pp. 103-104.

14. Hayhurst, Kelly J., C. Michael Holloway, "Challenges in Software Aspects of Aviation Systems," *Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop*, 27-29 November 2001, Greenbelt, MD, pp. 7-13.

15. Information Processing Ltd., "Advanced Coverage Metrics for Object-Oriented Software", available at http://www.iplbath.com/pdf/p0833.pdf. Visited on 30 October 2003.

16. FAA Aircraft Certification Service, Conducting Software Reviews Prior to Certification, Job Aid, June 1998, available at http://www2.faa.gov/certification/aircraft/av-info/software/Job_Aids.htm. Visited on 29 July 2003.

17. Knight, J.; Evans, D.; and Offutt, J.: Object Oriented Programming in Safety-Critical Software. A white paper.

18. Hanks, Kimberly S., John C. Knight, Elisabeth A. Strunk, "Erroneous Requirements: A Linguistic Basis for Their Occurrence and an Approach to Their Reduction," *Proceedings of the 26th Annual NASA Goddard Software Engineering Workshop*, 27-29 November 2001, Greenbelt, MD, pp. 115-119.

19. Knight, John C., Object-Oriented Techniques and Dependability, white paper.

20. Leveson, Nancy, "Re: object-orientation vs. safety-critical" in Safety-Critical Mailing List, 2002, archived at http://www.cs.york.ac.uk/hise/safety-critical-archive/2002/0203.html. Visited on 28 July 2003.

21. Liskov, Barbara H., Jeanette M. Wing: A Behavioral Notion of Subtyping, *ACM Transactions on Programming Languages and Systems*, Nov. 1994, vol. 16, no. 6, pp. 1811-1841.

22. Rierson, Leanna: "Object-Oriented Technology (OOT) in Civil Aviation Projects: Certification Concerns," *Proceedings of the 18th Digital Avionics Systems Conference*, St. Louis, MO, Oct. 24-29, 1999.

23. The Memory Management Reference Beginner's Guide Overview, archived at http://www.memorymanagement.org/articles/begin.html. Visited on 28 August 2003.

24. Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

25. Moynihan, Tony, 1996, "An Experimental Comparison of Object-Orientation and Functional-Decomposition as Paradigms for Communicating System Functionality to Users," *The Journal of Systems and Software*, vol. 33, pp. 163-169.

26. Offutt, Jeff, Roger Alexander, Ye Wu, Quansheng Xiao, Chuck Hutchinson, November 2001, "A Fault Model for Subtype Inheritance and Polymorphism," *The 12th IEEE International Symposium on Software Reliability Engineering*, Hong Kong, PRC, pp. 84–95.

27. Object Management Group, March 2003, OMG Unified Modeling Language Specification, Version 1.5, formal/03-03-01.

28. Rierson, Leanna K., FAA's Next Steps for OOTiA, presented at the Object Oriented Technology in Aviation Workshop 2, 27 March 2003, available at http://shemesh.larc.nasa.gov/foot/next-steps-end.ppt. Visited on 29 July 2003.

29. Rosay, Cyrille, Is DO-178B still compatible with modern modeling methods?, white paper from JAA/CEAT, draft 3-15, Friday 22 February 2002.

30. Hayhurst, Kelly J; Cheryl Dorsey; John Knight, Nancy Leveson, G. Frank McCormick, August 1999, Streamlining Software Aspects of Certification: Report on the SSAC Survey, NASA/TM-1999-209519.

31. Vessey, Iris, and Sue A. Conger, "Requirements Specification: Learning Object, Process, and Data Methodologies," *Communications of the ACM*, vol. 37, no. 5, May 1994, pp. 102-113.

32. Webster, Bruce F., *Pitfalls of Object-Oriented Development*, M&T Books, New York, New York, 1995. (out of print)

33. Whitford, S. A., Software Safety Code Analysis of an Embedded C++ Application, *Proceedings of the 20th International System Safety Conference*, Denver, CO, August 5-9, 2002, pp. 422-429.

34. Wood, M, J. Daly, J. Miller, and M. Roper, "Multi-Method Research: An Empirical Investigation of Object-Oriented Technology," *The Journal of Systems and Software*, 1999, no. 34, pp. 13-26.

Appendix A Results of the Beyond the Handbook Session

At OOTiA Workshop 2, a brainstorming session called "Beyond the Handbook" was held where participants were asked to suggest questions that should be answered *before* a program commits to using OOT. During the session, participants produced a list of fifty-one questions related to making a decision about whether to use OOT. The questions are listed below in the order recorded during the session.

- What is the impact of the handbook on choosing to use OOT?
- What are my alternatives to OOT
- What is the maturity of the staff experience in using OO methods?
- What is the maturity of the standards supporting the process?
- Does the criticality level affect the decision to use OOT?
- Is the issue with OOT the source to object code issue?
- How do specific languages relate to the guidelines in the handbook?
- Is there any evidence to suggest that we can build systems better with OOT?
- What is the real motivation for considering OOT? Is it to be faster, better, cheaper, or because programmers want to use Java?
- Can we teach programmers to build safe OO software? Is it being done anywhere?
- Can we analyse OO Software?
- Can we adequately test OO Software?
- Where's the data to support or deny OO claims?
- Is choosing technologies based on available pool of people negligence?
- What's the overall size and complexity of our project?
- What's the anticipated length of time our project must be maintained for?
- Is the company planning on having a product line architecture? How frequent will releases be?
- Does the cert authority have a track record for approval or denial of OO projects?
- What is the minimum experience a company should have in OOT before embarking on developing a system with major consequences?
- If I choose to use OO analysis and design but implement in a procedural language, what implication will the handbook have on me?
- Does the company have a plan for reuse?
- Is use-case, iterative requirements development good enough for safety critical systems?
- Is there a set of lessons learnt that the group should know?
- Does the company have established standards and practices for OOT?
- Where's the technology argument that we can build safe systems using OOT?
- To what extent is OOT unique? How do we determine the error cases that are unique to OOT?
- Whether your DER is OO literate or OO friendly?

- Is it appropriate to use commercially available processes/products for developing aviation software, or what steps need to be taken to make it so?
- Are we subjecting OOT to extra scrutiny because it is new to aviation community?
- Does the OO paradigm actually fit your problem domain?
- How much of current good practice is non-applicable to OOT?
- Do we need to make a distinction between the design process and the develop/test process? Does OOD <-> OO implementation?
- Do you have a plan for failure?
- Where is dynamic dispatch really useful? Would the non-OO alternative be any easier to analyse?
- How mature are your requirements?
- Is there an existing tool set to support your effort?
- Has the company analysed the benefits & risks of using OOT over their current established approaches to software engineering?
- Have you run out of steam in regards to existing techniques for really large systems and if so does OOT help us manage such systems better?
- Is there an agreement on what is OOT? What is essential, what is not etc.?
- Does OOT help us write the requirements correctly and implement them properly? Can we integrate Formal Methods into the process?
- Does OOT help us identify what we really want in the system and document this in requirements?
- Will our system interface with other systems that are not OOT based?
- Do you understand the issues presented in the handbook and why are they issues? Does anyone?
- Can your company make a case for using OOT in a system in such a way that you can document and verify the system?
- Why is company X using OOT?
- Can the system & software safety assessments be derived easily from OOT or is it a difficult task to ensure safety?
- Why is company Y not using OOT?
- Is OO the best method from the engineering point of view for reuse?
- Are the objectives in DO-178B sufficient to define ‘correctly’ for OOT development?
- Are your requirements implementable using OOT?
- What measures or metrics will you use to determine success/failure for your project?
- Is control-flow analysis, data-flow analysis, Z-flow analysis applicable to OO software, or should we be looking for something else?

Appendix B Mapping of Issue List to Considerations

The following table contains all of the entries[4], as of the date of publication of this document, to the Issues and Comments about Object Oriented Technology in Aviation list on the OOTiA web site. Four of the entries (IL 14, 36, 39, and 41) are duplicates of other entries and are denoted by an *. In the table, most entries are mapped to a DO-178B life cycle process category, and to one of the subjects of key concern discussed within that category. Some entries were difficult to classify and were not assigned to subject categories.

IL #	Issue Statement	DO-178B Life Cycle Process Impacted	Subject
1	Deactivated Code will be found in any application that uses general purposed libraries or object-oriented frameworks. (Note that this is the case where unused code is NOT removed by smart linkers.)	Development	2.3.2.4.2 libraries & frameworks
2	Flow Analysis, recommended for Levels A-C, is complicated by Dynamic Dispatch (just which method in the inheritance hierarchy is going to be called?).	Integral (Verification)	2.3.3.1.1 flow analysis
3	Timing Analysis, recommended for Levels A-D is complicated by Dynamic Dispatch (just how much time will be expended determining which method to call?).	Integral (Verification)	2.3.3.1.3 timing and stack analysis
4	Requirements Testing, recommended for Levels A-D, and Structural Coverage Analysis, recommended for Levels A-C, are complicated by Inheritance, Overriding and Dynamic Dispatch (just how much of the existing verification of the parent class can be reused in its subclasses?) [Also, inheritance and overriding raise concern with respect to testing (e.g., it is unclear whether superclass tests and test results for inherited and overridden functions may be reused). [3]]	Integral (Verification)	2.3.3.2.2 test case reuse
5	Structural Coverage Analysis, recommended for Levels A-C, is complicated by Dynamic Dispatch (just which method in the inheritance hierarchy does the execution apply to?). [A closely related issue is that inheritance and polymorphism may cause difficulty in obtaining structural coverage (both modified condition/decision coverage (MC/DC) and decision coverage) [3]]	Integral (Verification)	2.3.3.1.2 structural coverage analysis
6	Conformance to the guidelines in DO-178B concerning traceability from source code to object code for Level A software is complicated by Dynamic Dispatch (how is a dynamically dispatched call represented in the object code?). [inheritance and polymorphism may make source to object code correspondence difficult [3]]	Integral (Verification)	2.3.3.1.4 source to object trace
7	Polymorphic, dynamically bound messages can result in code that is error prone and hard to understand.	Development	2.3.2.2.1 unclear intent
8	Dynamic dispatch presents a problem with regard to the traceability of source code to object code that requires "additional verification" for level A systems as dictated by DO-178B section 6.4.4.2b.	Integral (Verification)	2.3.3.1.4 source to object trace
9	Dynamic dispatch complicates flow analysis, symbolic	Integral	2.3.3.1.1 flow

[4] The text contained in brackets […] at the end of an issue statement represents expansion/clarification of the issue from the cited source and, as such, will not be found in the original issue as stated on the web site.

	analysis, and structural coverage analysis.	(Verification)	analysis, 2.3.3.1.2 structural coverage analysis
10	Inheritance, polymorphism, and linkage can lead to ambiguity.	Development	2.3.2.2.1 unclear intent
11	The use of inheritance and polymorphism may cause difficulties in obtaining structural coverage, particularly decision coverage and MC/DC	Integral (Verification)	2.3.3.1.2 structural coverage analysis
12	Source to object code correspondence will vary between compilers for inheritance and polymorphism.	Integral (Verification)	2.3.3.1.4 source to object trace
13	Polymorphic and overloaded functions may make tracing and verifying the code difficult.	Integral (Traceability)	2.3.3.4.2 complexity
14*	Requirements Testing, recommended for Levels A-D, and Structural Coverage Analysis, recommended for Levels A-C, are complicated by Inheritance, Overriding and Dynamic Dispatch (just how much of the existing verification of the parent class can be reused in its subclasses?). *[Note: this is exactly the same issue as IL 4)]*	Integral (Verification)	2.3.3.2.2 test case reuse
15	Multiple interface inheritance can introduce cases in which the developer's intent is ambiguous. (when the same definition is inherited from more than one source is it intended to represent the same operation or a different one?)	Development	2.3.2.2.1 unclear intent
16	Flow Analysis and Structural Coverage Analysis, recommended for Levels A-C, are complicated by Multiple Implementation Inheritance (just which of the inherited implementations of a method is going to be called and which of the inherited implementations of an attribute is going to be referenced?). The situation is complicated by the fact that inherited elements may reference one another and interact in subtle ways which directly affect the behavior of the resulting system.	Integral (Verification)	2.3.3.1.1 flow analysis
17	Use of inheritance (either single or multiple) raises issues of compatibility between classes and subclasses.	Development	2.3.2.1.1 type substitutability
18	Inheritance and overriding raise a number of issues with respect to testing: "Should you retest inherited methods? Can you reuse superclass tests for inherited and overridden methods? To what extent should you exercise interaction among methods of all superclasses and of the subclass under test?"	Integral (Verification)	2.3.3.2.2 test case reuse
19	Inheritance can introduce problems related to initialization. "Deep class hierarchies [in particular] can lead to initialization bugs." There is also a risk that a subclass method will be called (via dynamic dispatch) by a higher level constructor before the attributes associated with the subclass have been initialized.	Development	2.3.2.3.2 initialization
20	"A subclass-specific implementation of a superclass method is [accidentally] omitted. As a result, that superclass method might be incorrectly bound to a subclass object, and a state could result that was valid for the superclass but invalid for the subclass owing to a stronger subclass invariant. For example, Object-level	Development	2.3.2.2.2 overriding

	methods like `isEqual` or copy are not overridden with a necessary subclass implementation".		
21	"A subclass [may be] incorrectly located in a hierarchy. For example, a developer locates SquareWindow as a subclass of RectangularWindow, reasoning that a square is a special case of a rectangle ... Suppose that [the method] resize(x, y) is inherited by SquareWindow. It allows different lengths for adjacent sides, which causes SquareWindow to fail after it has been resized. This situation is a design problem: a square is not a kind of a rectangle, or vice versa. Instead both are kinds of four-sided polygons. The corresponding design solution is a superclass FourSidedWindow, of which RectangularWindow and SquareWindow are subclasses."	Development	2.3.2.2.1 unclear intent
22	"A subclass either does not accept all messages that the superclass accepts or leaves the object in a state that is illegal in the superclass. This situation can occur in a hierarchy that should implement a subtype relationship that conforms to the Liskov substitution principle."	Development	2.3.2.1.1 type substitutability
23	"A subclass computes values that are not consistent with the superclass invariant or superclass state invariants."	Development	2.3.2.1.1 type substitutability
24	"Top-heavy multiple inheritance and very deep hierarchies (six or more subclasses) are error-prone, even when they conform to good design practice. The wrong variable type, variable, or method may be inherited, for example, due to confusion about a multiple inheritance structure"	Development	2.3.2.2.1 unclear intent
25	The ability of a subclass to directly reference inherited attributes tightly couples the definitions of the two classes. [Even when attributes are hidden from client classes, they may not be hidden from subclasses. [3]]	Development	2.3.2.2.1 unclear intent
26	Inheritance can be abused by using it as a "kind of code-sharing macro to support hacks without regard to the resulting semantics". [Inheritance can be misused to support code sharing without consideration of substitutability and the rules for behavioral subtyping (i.e., substitutability) [3] [26]]	Development	
27	When the same operation is inherited by an interface via more than one path through the interface hierarchy (repeated inheritance), it may be unclear whether this should result in a single operation in the subinterface, or in multiple operations.	Development	2.3.2.2.1 unclear intent
28	When a subinterface inherits different definitions of the same operation [as a result of redefinition along separate paths], it may be unclear whether/how they should be combined in the resulting subinterface.	Development	2.3.2.2.1 unclear intent
29	Use of multiple inheritance can lead to "name clashes" when more than one parent *independently* defines an operation with the same signature.	Development	2.3.2.2.1 unclear intent
30	When *different* parent interfaces define operations with different names but compatible specifications, it is unclear whether it should be possible to merge them in a subinterface.	Development	2.3.2.2.1 unclear intent
31	It is unclear whether the normal overload resolution rules	Development	2.3.2.2.2 overriding

	should apply between operations inherited from different superinterfaces or whether they should not (as in C++).		
32	It is important that the overriding of one operation by another and the joining of operations inherited from different sources always be intentional rather than accidental. [This is a concern with respect to both overriding of methods and attributes [26]]	Development	2.3.2.2.2 overriding
33	Multiple inheritance complicates the class hierarchy	Development	2.3.2.2.1 unclear intent
34	Multiple inheritance complicates configuration control	Integral (Configuration Management)	2.3.3.3.2 configuration control
35	When inheritance is used in the design, special care must be taken to maintain traceability. This is particularly a concern if multiple inheritance is used.	Integral (Traceability)	2.3.3.4.2 complexity
36*	Source to object code correspondence will vary between compilers for inheritance and polymorphism. *[Note: this is exactly the same issue as IL 12]*	Integral (Verification)	2.3.3.1.4 source to object trace
37	Overuse of inheritance, particularly multiple inheritance, can lead to unintended connections among classes, which could lead to difficulty in meeting the DO-178B/ED-12B objective of data and control coupling.	Development	2.3.2.2.1 unclear intent
38	Multiple inheritance should be avoided in safety critical, certified systems.	Planning	2.3.1.3 restrictions
39*	"Top-heavy multiple inheritance and very deep hierarchies (six or more subclasses) are error-prone, even when they conform to good design practice. The wrong variable type, variable, or method may be inherited, for example, due to confusion about a multiple inheritance structure" *[Note: this is exactly the same issue as IL 24]*	Development	2.3.2.2.1 unclear intent
40	Reliance on programmer specified optimizations of the inheritance hierarchy (invasive inheritance) is potentially error prone and unsuitable for safety critical applications.	Development	
41*	Inheritance, polymorphism, and linkage can lead to ambiguity. *[Note: this is exactly the same issue as IL 10]*	Development	2.3.2.2.1 unclear intent
42	Inheritance allows different objects to be treated in the same general way. Inheritance as used in Object Oriented Technology is combining several like things into a fundamental building block. The programmer is allowed to take a group of these like things and refer to them in a general way. One routine can be used for all types that inherit from the fundamental building block. The more often a programmer can use the generic behavior of the parent, the more productive the programmer is. The problem I see is that the generic behavior will not always be precise enough for all the applications, and that critical judgement is required to determine when the programmer needs to specialize the behavior of one of the object rather than use the generic. Who will issue that critical judgement? Who will find all the instances where the general case is too far away from the precision required?	Development	2.3.2.1.1 type substitutability
43	Flow Analysis, recommended for levels A-C, is impacted	Integral	2.3.3.1.1 flow

	by Inlining (just what are the data coupling and control coupling relationships in the executable code?). The data coupling and control coupling relationships can transfer from the inlined component to the inlining component.	(Verification)	analysis
44	Stack Usage and Timing Analysis, recommended for levels A-D, are impacted by Inlining (just what are the stack usage and worst-case timing relationships in the executable code?). Since inline expansion can eliminate parameter passing, this can effect the amount of information pushed on the stack as well as the total amount of code generated. This, in turn, can effect the stack usage and the timing analysis.	Integral (Verification)	2.3.3.1.3 timing and stack analysis
45	Structural Coverage Analysis, recommended for levels A-C, is complicated by Inlining (just what is the "logical" coverage of the inline expansions on the original source code?). This is generally only a problem when inlined code is optimized. If statements are removed from the inlined version of a component, then coverage of the inlined component is no longer sufficient to assert coverage of the original source code.	Integral (Verification)	2.3.3.1.2 structural coverage analysis
46	Conformance to the guidelines in DO-178B concerning traceability from source code to object code for Level A software is complicated by Inlining (is the object code traceable to the source code at all points of inlining/expansion?). Inline expansion may not be handled identically at different points of expansion. This can be especially true when inlined code is optimized.	Integral (Verification)	2.3.3.1.4 source to object trace
47	Inlining may affect tool usage and make structural coverage more difficult for levels A, B, and C.	Integral (Verification)	2.3.3.1.2 structural coverage analysis
48	The unrestricted use of certain object-oriented features may impact our ability to meet the structural coverage criteria of DO-178B.	Integral (Verification)	2.3.3.1.2 structural coverage analysis
49	Statement coverage when polymorphism, encapsulation or inheritance is used.	Integral (Verification)	2.3.3.1.2 structural coverage analysis
50	Templates are instantiated by substituting a specific type argument for each formal type parameter defined in the template class or operation. Passing a test suit for some but not all instantiations cannot guarantee that an untested instantiation is bug free.	Integral (Verification)s	2.3.3.1.3 timing and stack analysis
51	Nested templates, child packages (Ada), and friend classes (C++) can result in complex code and hard to read error messages on many compilers.	Development	2.3.2.2.1 unclear intent
52	Templates can be compiled using "code sharing" or "macro-expansion". Code sharing is highly parametric, with small changes in actual parameters resulting in dramatic differences in performance. Code coverage, therefore, is difficult and mappings from a generic unit to object code can be complex when the compiler uses the "code sharing" approach.	Integral (Verification)	2.3.3.1.2 structural coverage analysis
53	Macro-expansion can result in memory and timing issues, similar to those identified for inlining.	Integral (Verification)	2.3.3.1.3 timing and stack analysis

54	The use of templates can result in code bloat. Many C++ compilers cause object code to be repeated for each instance of a template of the same type.	Development	
55	How can we meet the structural coverage requirements of DO-178B with respect to dynamic dispatch? There is cause for concern because many current Structural Coverage Analysis tools do not "understand" dynamic dispatch, i.e. do not treat it as equivalent to a call to a dispatch routine containing a case statement that selects between alternative methods based on the run-time type of the object.	Integral (Verification)	2.3.3.1.2 structural coverage analysis
56	How can we meet the control and data flow analysis requirements of DO-178B with respect to dynamic dispatch?	Integral (Verification)	2.3.3.1.1 flow analysis
57	How can deactivated code be removed from an application when general purpose libraries and object-oriented frameworks are used but not all of the methods and attributes of the classes are needed by a particular application?	Development	2.3.2.4.2 libraries & frameworks
58	How can we enforce the rules that restrict the use of specific OO features?	Planning	2.3.1.3 restrictions
59	Implicit type conversion raises certification issues related to source to object code traceability, the potential loss of data or precision, and the ability to perform various forms of analysis called for by [DO-178B] including structural coverage analysis and data and control flow analysis. It may also introduce significant hidden overheads that affect the performance and timing of the application.	Integral (Verification)	2.3.3.1.4 source to object trace
60	Overloading can be confusing and contribute to human error when it introduces methods that have the same name but different semantics. Overloading can also complicate matters for tools (e.g., structural coverage and control flow analysis tools) if the overloading rules for the language are overly complex.	Development	2.3.2.2.2 overriding
61	Loss of traceability due to the translation of functional requirements to an object-oriented design.	Integral (Traceability)	2.3.3.4.1 function vs. object tracing
62	Functional coverage of the low level requirement	Integral (Verification)	2.3.3.2.1 requirements testing
63	Philosophy of Functional Software Engineering - Most of the training, tools and principles associated with software engineering and assurance, including those of RTCA DO-178B, have been focused on a software function perspective, in that there is an emphasis on software requirements and design and verification of those requirements and the resulting design using reviews, analyses, and requirements-based (functional) testing, and RBT coverage and structural coverage analysis. Philosophy of Objects and Operations - Although generally loosely and inconsistently defined, OOT focuses on "objects" and the "operations" performed by and/or to those objects, and may have a philosophy and perspective that are not very conducive to providing equivalent levels of design assurance as the current	Planning Development	2.3.1.2 requirements methods and notations

	"functional" approach.		
64	Software/software integration testing is often avoided. The position defended by the industry is that the high level of interaction between a great number of objects could lead to a combinative explosion of test cases.	Integral (Verification)	2.3.3.2.1 requirements testing
65	Could there be security concerns related to the use of COTS based OOT solutions? Particularly with respect to field loadable software, security risks have been mitigated by the unique architectures of most current systems.	Additional Considerations	
66	Use of dynamic memory allocation/deallocation and use of exception handling were raised as issues by Leanna Rierson in her paper "Object-Oriented Technology (OOT) in Civil Aviation Projects: Certification Concerns" but are currently missing from the list of concerns. If the FAA is concerned about these two items, they should be discussed at the workshop.	Development	2.3.2.3.1 indeterminate execution profiles
67	Most OO languages use reference semantics for passing objects (e.g. Java only supports reference semantics; C++ also supports passing by value but this is rarely used and cannot be used when dynamic binding is required). This results in variables being aliased to each other. It is difficult to analyse the effect of this aliasing on program behaviour because many tools do not allow for the possible presence of aliasing. it is also easy for a developer to inadvertantly use a shallow copy or equality operation where the required semantics can only be achieved by a deep copy or equality operation.	Integral (Verification)	
68	The selection of the code to implement an operation may depend upon more than just the run time type of the target object. In cases involving binary mathematical operations, for instance, this choice typically depends on the run time types of both arguments. As explained in [Bruce et al.], [Castagna] and [MultiJava], this (and other related situations) are not handled well by most current OO languages. (A.k.a. "Binary methods problem") References: [Bruce eta al.] Bruce, Kim, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens and Benjamin Pierce. On Binary Methods, Iowa State University, technical report #95-08a, December 1995. [Castagna] Castagna, Giuseppe. Object-Oriented Programming: A Unified Foundation, Birkauser, Boston, ISBN: 0-8176-3905-5, 1997. [MultiJava] Clifton, Curtis, Gary T. Leavens, Craig Chambers, and Todd Millstein. "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java", OOPSLA 2000 Conference Proceedings: ACM SIGPLAN Notices, vol. 35, no. 10, October 2000, pp. 130-145.	Development or Integral (Verification)?	
69	The use of OO methods typically leads to the creation of many small methods which are physically distributed over a large number of classes. This, and the use of dynamic dispatch, can make it difficult for developers to trace critical paths through the application during design and coding reviews.	Integral (Traceability)	2.3.3.4.1 function vs. object tracing

70	The difference between dead and deactivated code is not always clear when using OOT. Without good traceability, identifying dead vs. deactivated code may be difficult or impossible.	Development Integral (Traceability)	2.3.2.4.1 identifying dead and deactivated code
71	When a design contains abstract base classes, portions of the implementations of these classes may be overridden in more specialized subclasses, resulting deactivated code.	Development	2.3.2.4.1 identifying dead and deactivated code
72	Traceability is made more difficult because there is often a lack of OO methods or tools for the full software lifecycle.	Integral (Traceability)	2.3.3.4.3 tracing through OO views
73	Formal specification languages are generally accessible only to those specially trained to use them. To make formal specifications accessible to developers and the authors of test cases, we must map such formal specifications to natural language and/or other less formal notations (e.g. UML). There, however, is currently no well defined means of doing so. This issue applies to both preliminary and detailed design.	Planning	2.3.1.2 requirements methods and notations
74	Change impact analysis may be difficult or impossible due to difficulty in tracing functional requirements through implementation.	Integral (Configuration Management)	2.3.3.3.2 configuration control
75	Limitations of UML may limit how non-functional and cross-cutting requirements of realtime, safety critical, distributed, fault-tolerant, embedded systems are captured in UML and traced to the design, implementation, and test cases.	Planning Development	2.3.1.2 requirements methods and notations
76	Configuration management may be difficult in OO systems, causing traceability problems. If the objects and classes are considered configuration items, they can be difficult to trace, when used multiple times in slightly different manners.	Integral (Configuration Management)	2.3.3.3.1 configuration identification
77	What is "low level requirements" for OO? Affects how we do low-level testing. If we don't know what low-level requirements are, we don't know the appropriate level of testing. * High level = WHAT * Low level = HOW Related to issue raised in tools session – relation be between artifacts. Should be addressed in the handbook.	Planning	2.3.1.1 defining life cycle data
78	Addressing derived requirements for OO – how does this happen? How is it different than traditional and how does it tie up to the safety assessment. Not really unique for OO. Will be addressed when we do the artifact mapping.	Planning	2.3.1.2 requirements methods and notations
79	Difficult to identify individual atomic requirements in OO. UML tends to group requirements in a graphical	Planning	2.3.1.2 requirements methods and

	format. Would complicate matters if considered derived. For derived requirements, the entire graph would be passed to the safety folk for evaluation of safety impact.		notations
80	Lower levels of decomposition may not be possible for some requirements (e.g., performance requirements). Levels of abstraction may be different than traditional.	Planning Development	2.3.1.2 requirements methods and notations
81	Are there unique challenges for source to object code traceability in non-Level A systems? Where should this be addressed? Multiple tools and ways of addressing s-to-o traceability? (not really new) Beyond what DO-178B requires. More of a "DO-178C" issue. Out of scope for the handbook. Is UML the "source code" for OO?	Integral (Verification)	2.3.3.1.4 source to object trace
82	Is there another "class" of tool qualification for visual modeling tools to demonstrate the integrity of these tools? Not necessarily automating a step, but are looking to make sure the tool is doing what you want. How to ensure consistency of the tools (validating the tool)? How to validate the tool when changes occur? Typically part of the tool selection process. Concern seems to be addressed by handbook mod.	Additional Considerations	2.3.4.3 tool qualification
83	Auto-test and code generation tools – what are the concerns when a single tool generates code and test from the same model? The concern is with the independence – same input and same tool. Already covered by DO-178B. Not necessarily OO-specific, but may be more prevalent with OO tools. Need to be addressed in some other document or forum.	Additional Considerations	2.3.4.3 tool qualification
84	Maintaining tool environment, archives, … when licenses are involved is not clear. May need to have some kind of "permanent license" to support safety and continued airworthiness of the aircraft. OO more dependent on tools, but not necessarily an OO-specific issue.	Additional Considerations	2.3.4.2 tool environments
85	Maturity/long-term support of tools. Tool manufacturers may not realize the long-life need of tools. Is this a higher risk in the OO environment? Education for both the tool and aviation communities to understand the specific needs for tool manufacturers and aircraft manufacturers. Not necessarily OO-specific, but might be more prevalent with OO.	Additional Considerations	2.3.4.2 tool environments
86	Are there other types of OO tools that need to be addressed? Need to anticipate other classes of tools that may come onto the scene. E.g., traceability tool for OO, transformation tools, CM tools, refactoring tools (tool to restructure source code to meet new requirements),	Additional Considerations	2.3.4.2 tool environments

87	How does OO life cycle data map to the DO-178B section 11 life cycle data? E.g., What "source code" mean in OO? What is req, design, code? Transition from text-based to model-based artifacts. *** May need to clarify this up front in the handbook, when making the tie between DO-178B and the handbook.	Planning	2.3.1.1 defining life cycle data
88	Configuration management and incremental development of OO projects and tools. When CM comes into play during the development process may be different than our current practices, when using an UML tool. Doing more iterations in OO. How to "get credit" on iterations. Not necessarily OO-specific, but might be more prevalent with OO because of the multiple iterations.	Integral (Configuration Management)	2.3.3.3.2 configuration control
89	Is dynamic dispatch compatible with DO-178B required forms of static analysis? Mention that dynamic dispatch hinders some forms of static analysis including (see DO-178B section 6.3.4f). Tools can treat this if complete closure exists. DO-178B requires complete closure. In cases of incomplete closure, need to define ways to implement.	Integral (Verification)	2.3.3.1.1 flow analysis
90	Fundamental pre-requisite language issues need clarification prior to adopting LSP and DBC. How can LSP be implemented using available languages? Strongly consider a language subset that is amenable to use of LSP and DBC. Concern is how far to take this subset.	Development	2.3.2.1.1 type substitutability
91	Inconsistent Type Use (ITU): When a descendant class does not override any inherited method (i.e., no polymorphic behavior), anomalous behavior can occur if the descendant class has extension methods resulting in an inconsistent inherited state. [5]	Development	2.3.2.1.2 inconsistent type use
92	State Definition Anomaly (SDA): If refining methods do not provide definitions for inherited state variables that are consistent with definitions in an overridden method, a data flow anomaly can occur. [6]	Development	2.3.2.2.2 overriding
93	State Definition Inconsistency (SDIH): If an indiscriminately-named local state variable is introduced, a data flow anomaly can result.	Development	2.3.2.2.2 overriding
94	State Defined Incorrectly (SDI): If a computation performed by an overriding method is	Development	2.3.2.2.2 overriding

[5] Inconsistent Type Use (ITU): This is addressed by verification of subtyping (LSP). Where we assume that the meaning of "inconsistent state" is "violates the class invariant". [26]

[6] State Definition Anomaly (SDA): This goes beyond an initialization problem to LSP and breaking promises made by superclasses, i.e. by not keeping postconditions. [26]

	not semantically equivalent to the computation of the overridden method wrt a variable, a behavior anomaly can result. [7]		
95	Indirect Inconsistent State Definition (IISD): When a descendent adds an extension method that defines an inherited state variable, an inconsistent state definition can occur. [8]	Development	2.3.2.1.1 type substitutability
96	Anomalous construction behavior (ACB1): If a descendant class provides an overriding definition of a method which uses variables defined in the descendant's state space, a data flow anomaly can occur.[9]	Development	2.3.2.2.2 overriding
97	Anomalous construction behavior (ACB2): If a descendant class provides an overriding definition of a method which uses variables defined in the ancestor's state space, a data flow anomaly can occur. [10]	Development	2.3.2.2.2 overriding
98	Incomplete construction (IC): If the constructor does not establish initial state conditions and the state invariants for new instances of a class, then a state variable may have in incorrect initial value or a state variable may not have been initialized. [11]	Development	2.3.2.3.2 initialization
99	State Visibility Anomaly (SVA): When private state variables exist, if every overriding method in a descendant class doesn't call the overridden method in the ancestor class, a data flow anomaly can exist. [12]	Development	2.3.2.2.2 overriding
100	When using OO tools to develop software requirements, design and implementation, it is beneficial to work at the visual model level, especially when using UML. When working with OO tools, configuration management might be done at the modeling level (i.e., diagrams). This may cause a concern when the OO tools can introduce subtle errors into the diagrams.	Additional Considerations	2.3.4.3 tool qualification
101	Current visual modeling tools that are used for OO development make use of frameworks for automatic code generation, replacing tedious programming tasks. Frameworks may include patterns, templates, generics, and classes in ways requiring new verification approaches. The tool's framework may or may not enforce requirements, design and coding standards.	Additional Considerations	2.3.4.1 tool capability
102	Current visual modeling tools that are used for OO development provide a capability to generate source code	Additional Considerations	2.3.4.1 tool capability

[7] State Defined Incorrectly (SDI): Violates LSP in that promises to clients by superclasses are not kept. "Incorrect" means either breaking an invariant or breaking a postcondition by defining an incorrect "v" (per example). [26]

[8] Indirect Inconsistent State Definition (IISD): Violations of this rule represent violations of substitutability in that promises to clients by superclasses are not kept. As a reulst, IISD can be resolved by verification of substitutability, where inconsistent state means breaking the invariant, and methods added must respect invariants, e.g., the invariant for X must be same as or stronger than W in [26].

[9] Anomalous construction behavior (ACB1): Relates to the use of dynamic dispatch during initialization, which the initialization rule forbids. [26]

[10] Anomalous construction behavior (ACB2): A variation on the previous issue that is dealt with in the same way. [26]

[11] Incomplete construction (IC): Constructors must initialize (define) all attributes (variables). [26]

[12] State Visibility Anomaly (SVA): Per the example given in [26], B::m() must be compatible with both A::m() and C::m(). When introducing a method B:m() that both overrides a method A::m() and is overridden by another method C::m(), compatibility must be compatible in both directions. [26]

	directly from UML models. Most of the existing UML tools today can use visual modeling diagrams to construct models and generate source code from these models. The level of source code generation depends on the tool and on the user of the tool. It is unclear how such tools may be used in aviation projects.		
103	The current structural coverage tools available may not "be aware" or have visibility to the internals of inherited methods and attributes and polymorphic references supported with dynamic binding such that they can provide a reliable measurement of the structural coverage achieved by the requirements-based testing.	Additional Considerations	2.3.4.1 tool capability
104	Class hierarchies can become overly complex, which complicates traceability. Generalization, weak aggregation, strong aggregation, association and composition are some of the relations that can be used to create the class diagrams.	Integral (Traceability)	2.3.3.4.2 complexity
105	Iterative development is often desired in OO implementation. Each iterative cycle has its own requirements (normally a set of Use Cases), design, implementation, and test. There is a risk of losing traceability when using iterative development. This can be caused by adding or changing requirements, design, or implementations.	Integral (Traceability)	2.3.3.4.4 iterative development
106	Reusability is one of the objectives of OO development, but reusable components may be hard to trace because they are designed to support multiple usages of the same component. Reusable components may also have functionality that may not be used in every application.	Development	2.3.2.4.1 identifying dead and deactivated code
107	If polymorphism and dynamic binding are implemented, this can cause the stack size to grow, making it difficult to analyze the optimal stack size.	Integral (Verification)	2.3.3.1.3 timing and stack analysis

Appendix C Additional Considerations for Project Planning

In an early phase of the OOTiA project, several pitfalls identified by Webster in *Pitfalls of Object-Oriented Development* were proposed as helpful considerations in planning on OO program. These pitfalls were not identified through the data collection process as part of the OOTiA program, but were thought by the FAA to be useful. Because Webster's book is out of print, a number of ideas from the book are described below.

Conceptual Pitfalls

***Going object-oriented for the wrong reasons*:** There are "many more misunderstood, misguided, or just downright bad reasons for going object-oriented" [32]. These include (but are not limited to): wanting to cut back on development staff, thinking OOT will significantly reduce the need for testing, thinking that your project can be completed five to ten times faster using OOT, thinking the use of OOT will reduce project risk, and wanting to be able to build future products simply by plugging software components together.

***Thinking objects will solve all problems*:** As Fred Brooks has declared, and experience has repeatedly shown, there is no "silver bullet" to slay the problems associated with software development. A variety of techniques are needed. OOT can be a significant part of this process, but it will not (in and of itself) resolve most of the problems. Also, OOT does not come for free (in terms of the need for experience, training, and tools).

***Allowing new features to creep (or pour) in*:** The temptation to add new features certainly isn't unique to the use of OOT. The problem is that OOT (by isolating likely future changes) makes it easier to introduce such features. To avoid related problems, new features should generally be introduced only in new increments, as part of a planning process that takes into account cost and schedule.

Analysis and Design Pitfalls

***Pouring new wine into old bottles (or vice versa)*:** Problems can arise when OO and non-OO approaches are mixed with regard to development of the same software. For instance, a classic approach to analysis and design may be implemented using OOT techniques. Or an object-oriented analysis and design may be implemented without support for classes, class hierarchies, etc. In both cases, the result is often objects that resemble "slightly intelligent data structures, or procedural libraries" [32], and problems representing the relationships between classes. It is far better to divide an application into subsystems that either use or do not use OOT, then integrate the overall software system using wrappers to provide the needed subsystem interfaces in the proper style.

Class and Object Pitfalls

***Confusing subtyping with association*:** There are two basic relationships between classes. The subtyping relationship (inheritance) indicates that all instances of one class are logically instances of another. For example, an Airspeed Tape is a subtype of Tape, a GPS Sensor is a subtype of Sensor, etc. This relationship requires that we follow appropriate rules for subtyping. An association (often implemented using pointers) specifies that an instance of one class is linked to some number of instances of another class. For example, a car has an engine and is owned by some person. An engine is not logically a type of car, nor is a person. Relationships between cars, engines, and owners may also change at run time. Although the distinction between these two types of relationships seems obvious, novice OO developers often confuse the two, or misuse implementation inheritance to represent associations. This leads to problems, typically discovered during testing, that are far easier to avoid than to correct.

***Converting non-object code straight into objects*:** Developers with no object-oriented experience are used to treating data and functions (processes) separately. Data is organized into records or structures that may be linked via references or pointers. Functions are decomposed into smaller functions that perform a subset of the steps required by their parent. OOT combines the data oriented and function oriented approaches, organizing the program around collections of data (records) and the sets of functions that manipulate them. "Developers new to OO sometimes extend their former approaches in a one-sided fashion. For example, they may take a data structure, turn it into a class, define methods to access (get and set) the data fields, and leave it at that. The resulting class is little more than a cumbersome version of the original data structure, with no intelligence or behavior built into it. On the process side, developers may take a set of functions and turn them into methods of a single class, possibly with few no data members." [32] Both represent a misuse of OO principles, and lead to poor designs.

Verification Pitfalls

***Neglecting component testing*:** Although it is possible to take a subsystem or system level testing approach to an OO system, there are advantages to testing individual components. These components may be individual classes or larger units (assemblies of classes representing libraries or subsystems). They, however, should correspond to reusable entities. Testing at this level makes it easier to enforce the principles of Design by Contract [24], makes it easier to "inherit" test cases, and makes it possible to deliver individual components, their test cases, documentation, etc. as a single package.

Reuse Pitfalls

***Having or setting unrealistic expectations*:** OOT is often sold on promises related to reuse. Reuse, however, is not easy, and is not free. It comes at a cost (in terms of analysis and design) that many organizations are unwilling to pay. It also may only make economic sense if the organization plans to build three or more additional systems that are closely related to one another (form a product family), or if an individual component or subsystem can be reused at least three times.

***Being too focused on code reuse*:** "The most important creation to come from an OOT project is often the architecture and the design, not the code implementation." [32] This is true because these artifacts are typically more general, and more easily applied to new systems and new applications.